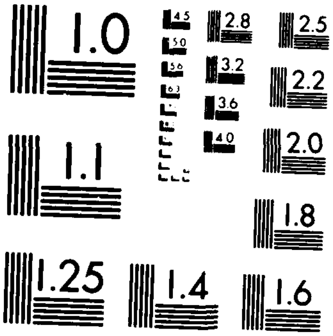
AD-A188 872 DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE 1/2
DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB
OH C W STANSBERRY DEC 87 AFIT/GCS/ENG/87D-26

UNCLASSIFIED F/G 12/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

1

DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE DESIGN

THESIS

Charles Wayne Stansberry, Jr.
Captain, USAF

AFIT/GCS/ENG/87D-26

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCS/ENG/87D-26

DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE DESIGN

THESIS

Charles Wayne Stansberry, Jr.
Captain, USAF

AFIT/GCS/ENG/87D-26

Approved for public release; distribution unlimited

AFIT/GCS/ENG/87D-26

DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE DESIGN

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Information Systems)

Charles Wayne Stansberry, Jr., B.S.
Captain, USAF

December, 1987

Approved for public release; distribution unlimited

Preface

The purpose of this thesis was to design and implement a computer tool which automates algorithms and functions which are used to design and study the logical structures of relational databases. Computer assistance is needed in this area because much of the current dependency theory used to design and study the logical structures of relational databases exists in the form of published algorithms and theorems, and hand simulating these algorithms can be a tedious and error prone chore. Additionally, since the process of logical design can be time consuming, repetitive, and can be structured into a well defined set of steps, it is well suited for computer assistance.

The computer tool, or "Dependency Theory Toolbox", was designed for use in an academic environment as a teaching aid and research tool, rather than for practical application to database design problems. The toolbox provides many functions which allow the user to generate and study database designs, and is specifically designed to support research in the area of alternative database designs. Much research is still needed in this area to define methods for automatically generating alternative designs.

Throughout this thesis effort, I have had a great deal of help from others. This project would not have been possible without the insights and assistance provided by my thesis advisor, Capt Mark A. Roth. His thorough knowledge of the subject area, and his insightful guidance significantly influenced the direction of the project, and enabled the thesis project to progress at a steady pace to completion. Additionally, I would like to thank the members of my thesis committee, Dr. Thomas C. Hartrum and Dr. Gary B. Lamont. Their assistance and suggestions were very helpful, and lead to many improvements in the final product. Finally, I would like to thank my wife, Cheryl, for her understanding and support throughout the project.

Charles Wayne Stansberry, Jr.

Table of Contents

Page

Page

List of Figures

List of Tables

AFIT/GCS/ENG/87D-26

Abstract

Many of the key concepts used for the logical design of relational databases are based on "dependency theory". In dependency theory, database dependencies are used to specify the constraints which must hold on a database. Dependency theory studies the properties of these dependencies and their use in the logical design of relational database systems.

Much of the current dependency theory used to design and study relational databases exists in the form of published algorithms and theorems. However, hand simulating these algorithms can be a tedious and error prone chore. Therefore, the purpose of this thesis investigation was to design and implement a computer tool (that is, a "toolbox") which contains various relational database design algorithms and functions to help solve the problems created by hand simulating the algorithms.

To establish the basis for additional work in the area of computer assisted database design, and to determine which algorithms and functions should be implemented in the toolbox, this thesis includes a review of the activities typically done to design a relational database, and surveys the computer tools which are available, or are being developed, to assist database designers with the logical design of relational databases. The survey of computer tools indicated that although many researchers have developed computer tools to assist with relational database design, there are still many algorithms and functions which need to be incorporated into automated design tools.

The toolbox implements algorithms to accomplish the following functions: 3NF decomposition, 4NF decomposition, BCNF decomposition, envelope set, FD/MVD minimal cover, dependency basis, minimal cover, membership algorithm, attribute closure, Armstrong relation instance, and support for generation of alternative logical designs. A simple menu-driven interface was created to access the toolbox functions.

The thesis provides an overview of the dependency theory concepts and definitions which are pertinent to understanding the algorithms and functions. Additionally, the thesis includes a discussion of how the decomposition algorithms can be used to generate alternative designs by changing the order of the dependencies in the input set, and by varying the order of the attributes on the left hand side of the dependencies. The toolbox includes functions to support research in this area.

The toolbox is intended for use in an academic environment as a teaching aid and research tool rather than for practical application to database design problems. However, the tool could be used to design small relational databases which have a limited number of attributes. An evaluation of the toolbox done during the acceptance testing phase of development indicates that the tool can effectively serve in all of these capacities.

DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE DESIGN

I. Introduction

1.1 Background

Designing a database is very time consuming and consists of a complex set of activities. The design process includes all the activities associated with analyzing, collecting, and organizing data into both logical and physical structures which can be implemented on a computer. The logical database structures are the file structures which show what the data is, and how it is interrelated [10]. These structures are not dependent on the specific computer on which they will be implemented. In the relational database model, the logical structures are a set of relations (tables) which contain all of the data elements (attributes) for the database. The physical database structures are the structures which must comply with the specifications and considerations of the specific computer on which they will be implemented. If these structures are not designed properly, the database may not perform efficiently, or may not consistently maintain the integrity of the data stored in it. Therefore, in order to design efficient and effective databases, the designer must not only incorporate all important information into the database, but he must also determine the best logical and physical structures for the data.

To ensure all important information is incorporated into the database, the designer may follow one of the many structured design methodologies outlined in the literature. Several of these methodologies, described in Chapter II, provide an overview of the tasks involved in database design. In general, most of the structured methodologies divide the design process into distinct phases to provide a systematic and thorough approach to the design problem.

The designer may approach the logical design of a relational database several different ways. For example, the structures may be designed by developing a conceptual model of the database in, for example, an Entity-Relationship (ER) diagram, and then directly mapping the ER diagram into a set of corresponding tables [16]. Another way to design the logical structures is to collect all necessary database attributes and database constraints (database dependencies), and then hand simulate design algorithms which generate the logical structures. Alternatively, the designer may use one of the many computer aided database design tools which have been developed by academic researchers and private companies. Many of the available tools are reviewed in Chapter II to provide an overview of the computer tools which are currently available to assist database designers.

Although researchers have investigated implementing computer tools to assist designers in all phases of database design, a significant effort has been applied in the area of logical design. This is because the process of logical design is well suited for computer assistance since it can be time consuming, repetitive, and can be structured into a clearly defined set of steps.

Many of the key concepts in logical design of relational databases are based on "dependency theory". In dependency theory, database dependencies are used as a type of language to specify the constraints which must hold on a database [12]. "Dependency theory studies the properties of this language and its use in database management systems" [12, page 19]. The majority of the computer tools which support the logical design process implement algorithms and functions which design logical structures by manipulating data dependencies and organizing the data elements into forms with certain desirable properties. These database forms are known as "normal" forms, and they are desirable because, among other things, they ensure minimal redundancy of data, ensure that no information is lost when the data is organized into separate relations, and they help avoid anomalies which can sometimes occur when updating unnormalized databases. Dependency theory and normalization are discussed in more detail in Chapter II.

Although many researchers have developed computer tools to assist database designers with logical design of relational databases, there are still many algorithms and functions which need to be incorporated into automated design tools. Since computer aided design tools can greatly enhance the efficiency and effectiveness of accomplishing database design tasks, the existence of these tools can support the development of better database designs, and can also assist students who are studying the process of database design.

1.2 Description of Problem

Much of the current dependency theory used to design and study relational databases exists in the form of published algorithms and theorems. Hand simulating these algorithms can be a tedious and error prone chore. Therefore, a toolbox of algorithms and functions to manipulate database dependencies would be beneficial in exploring the behavior of these algorithms, in assisting with the development of new algorithms, and in the active use of database design.

Thus, many database design algorithms should be implemented in a computer aided design tool (that is, a "toolbox") so that database designers and students studying database design have access to automated tools to assist their efforts. The toolbox should include algorithms which generate logical structures in specific normal forms, and also, algorithms which show alternative designs. A toolbox which automates the execution of database design algorithms such as these would greatly enhance the ability of designers and students to use the algorithms effectively.

Thus, the purpose of this thesis project was to implement a toolbox of algorithms and functions to assist database designers and students studying database design. This computer tool will help to solve the problems created by hand simulating the design algorithms.

1.3 Scope

The three main models used to define the data stored in a database include the hierarchical model, the network model, and the relational model [16]. This thesis effort was limited to developing a computer aided design tool which supports the relational model. The relational model was chosen because it has widespread acceptance and is therefore the subject of much of the current database research.

Additionally, although automated tools can be useful for all phases of database design, this investigation only involved implementation of a tool to assist with the logical design phase. The logical design phase was chosen because organizing data into tables which have the specific properties required to optimize database manipulation and data integrity requires that the database designer use algorithms and functions which can be very tedious to simulate manually. Additionally, much of the database course work done at the Air Force Institute of Technology focuses on the logical design of databases, and therefore, an automated tool which supports this phase of database design could be useful to the students at the Institute.

The use of this tool will be mainly pedagogical. That is, it is intended for use in an academic environment as a teaching aid and research tool rather than for practical application to database design problems. However, the tool could be used to design small relational databases which have a limited number of attributes.

This thesis effort did not involve proving the correctness of database design algorithms and functions. The algorithms and functions implemented in the computer tool will be extracted from published material, and the users of the tool will be referred to the original publication for the proof of correctness if it exists.

1.4 Approach

The "Dependency Theory Toolbox for Database Design" was developed using a standard phased approach to software development. The development phases included: Requirements Analysis, Design, Coding and Implementation, and Acceptance Testing. Before designing the system, an extensive literature review was accomplished to determine what work had already been done in the area of computer aided database design. The literature review was then used as an input to the first phase of system development to help establish the requirements for the toolbox.

In the first phase, the requirements for the toolbox were determined. This included defining the system objectives, the system user, the functional requirements of the system, and the database design algorithms required to implement the functions. Then, in the second phase, design issues were examined, system modules were planned, data and file structures were chosen, and the user interface was designed. Next, in the third phase, the system was coded and implemented. And finally, in the last phase, the system was tested to ensure it operated properly.

1.5 Sequence of Presentation

The remainder of this thesis is organized to document the activities accomplished in each phase of the development process. Chapter II contains a summary of the current knowledge related to computer aided database design tools, and reviews the pertinent concepts of dependency theory and normalization. Chapter III outlines the system requirements established during the requirements analysis phase. Then, Chapter IV documents the design phase, Chapter V documents the coding and implementation phase, and Chapter VI documents the acceptance testing phase. The last chapter, Chapter VII, presents conclusions and recommendations for further study. Additional system documentation is provided in the appendices.

II. Summary of Current Knowledge

The process of database design can be divided into several phases. Typically, these phases include the requirements collection and analysis phase, the conceptual design phase, the logical design phase, and the physical design phase [2]. Researchers have investigated implementing Computer Aided Design tools to assist database designers in all of these phases. In order to establish the basis for additional work in the area of computer assisted database design, this chapter summarizes the activities typically done to design a database, and then surveys the computer tools which are available, or being developed, to assist database designers with logical design of relational databases.

In the first section, this chapter reviews some of the structured design methodologies which database designers may use to ensure all important information is included in the database. Then, the next section provides an overview of "dependency theory" and "normalization" which are the basis for many of the key concepts in logical design of relational databases. The third section then surveys the computer tools developed for logical database design, and the final section presents a summary of the information.

2.1 Database Design Methodologies

To help ensure all important information is incorporated into the database, the designer can follow one of the many structured database design methodologies which have been outlined in the current literature [2,6,9,24]. These structured methodologies provide a systematic approach to database design, and usually divide the design process into several distinct phases. One such methodology is described by Batini, et al. [2]. The phases defined for this approach include the Requirements Collection and Analysis phase, the Conceptual Design phase, the Logical Design phase, and the Physical Design phase. The authors outline the activities accomplished in each phase as follows.

The Requirements Collection and Analysis phase consists of the activities required to collect the requirements for data, operations, and events from the user. The user supplies the requirements in plain language sentences. Then, the database designers translate the requirements into a more precise language using different sentence types to describe data, operations, and events.

The next phase, the Conceptual Design phase, includes the activities required to formalize the description of the data, the operations, and the events which were collected during the first phase. During the Conceptual Design phase, the data is organized by using Entity Relationship (ER) diagrams, which are graphical representations of the data elements (entities), the characteristics (attributes) of the entities, and the relationships among those entities. This process is an incremental process of choosing all of the entities required for a specific operation, and then defining the ER diagram for just that specific portion of the data elements. The resulting diagram is called an operation schema. The data schema is defined as all of the data which are required to support the operations on the database. So, as each operation schema is defined, it is integrated into the overall data schema. Thus, a data schema is developed for each user's view (perspective), with the data they need for their particular operations, and then the separate views are merged into a global view which contains all of the data elements in the entire organization's database.

The next phase of design, the Logical Design phase, is defined as the activities which are required to translate the conceptual model created in the previous phase into a logical model. The logical model for a relational database consists of a set of relations (tables) which contain all of the data elements for the database. The tables are set up so that they contain related data.

The Physical Design phase consists of determining how the data will actually be stored in the memory of the target computer. For relational databases, this phase consists mainly of determining what type of indexing system should be used to access the relations. The type of indexing should be chosen to minimize the costs of operations on the database [2]. Thus, the overall methodology described by Batini, et al., considers all of the major activities involved in designing a database, and provides a well defined structured approach to the design process.

Other authors have defined similar database design methodologies. For example, Herman outlined a design methodology which also consists of four phases but with slightly different names and activities within the phases [14]. Herman defines the four phases of database design as the Conceptual Design phase, the Detailed Conceptual Design phase, the Logical Design phase, and the Physical Design phase. These four phases correspond to the standard phases of the software project life cycle: Feasibility, Function Analysis, Design, and Implementation. Thus, this design methodology has an appeal in environments where structured programming and design techniques are already being used [14]. Like the design method defined by Batini, et al., the method defined by Herman also considers all of the major activities involved in designing a database, and provides a well defined structured approach to the database design process.

A third database design methodology, again similar to those already discussed, is outlined in the article, "The Database Design and Evaluation Workbench (DDEW) at CCA" by Reiner and others [24]. Thus, we can see that several structured approaches to database design have been defined in the literature, and that most of the approaches clearly define steps or phases to follow in the design process. Also, since the process of database design is very complex and time consuming, these structured design approaches should help to ensure that all important information is considered and ultimately incorporated into the database design [15].

Since many of the tasks involved in designing a database are time consuming and repetitive, they are good candidates for Computer Aided Design (CAD) tools [19]. Additionally, as Bjornerstedt and Hulten said, "Systems for managing large scale databases under the relational model have become commercially available and therefore the value of design tools for the relational model is obvious" [6, page 215]. Thus, to enhance the database design process, many researchers have developed computer tools to aid database designers.

Although researchers have investigated implementing CAD tools to assist designers in all phases of database design, a significant effort has been applied in the area of logical design. This is because the process of logical design is well suited for computer assistance because the process can be time consuming, repetitive, and it can be structured into a clearly defined set of steps. Many of the key concepts in logical design of relational databases are based on "dependency theory" and "normalization". The next section provides an overview of these two areas.

2.2 Dependency Theory and Normalization

The activities required for designing the logical structures of a relational database have been extensively documented in the area of relational database theory known as dependency theory. As stated in Chapter I, in dependency theory, database dependencies are used as a type of language to specify the constraints which must hold on a database [12]. "Dependency theory studies the properties of this language and its use in database management systems" [12, page 19].

There are many types of data dependencies, including functional dependencies, multivalued dependencies, join dependencies, inclusion dependencies, etc. See [12] for a complete survey. Although all of these types of data dependencies are useful for completely defining the semantics of a database (by semantics, we mean all the constraints which must hold for the entire database), this paper focuses on functional and multivalued dependencies because both types have been used effectively to organize the database elements (attributes) into relations in certain "normal" forms. The process of organizing the attributes of a database into relations in a certain normal form is called "normalization". In general, the objective of normalization is to organize, or decompose, the attributes into relations which minimize repetition of data, and which allow easy retrieval of required information [16]. Additionally, normal forms help avoid certain types of anomalies which occur in database manipulation [8].

Many normal forms have been defined in the literature, including:

1. First Normal Form (1NF)
2. Second Normal Form (2NF)
3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)

A good description of all of the above normal forms can be found in [13]. A major goal of the normalization process is to generate relations which are "lossless" and "dependency preserving" [16]. "Lossless" means that relations must also be available in the decomposed set of relations [8], or it must be possible to retrieve the information by joining relations together. "Dependency preserving" means that the attributes have been grouped into relations so that it is not necessary to join relations to verify whether a certain integrity constraint (i.e., a dependency) has been violated. That is, all the attributes of each dependency must be embedded in single relations, or implied by dependencies which are embedded in the relations, so it is not necessary to compute joins to verify whether the integrity of the database has been violated [16].

Normalization can be accomplished using functional dependencies (FDs) and multivalued dependencies (MVDs) either individually or together. Functional dependencies are a type of constraint of the form "X → Y", read as "X determines Y", where X and Y are attributes in the database. Informally, this means that if a relation in the database contains columns for both X and Y, the value of Y is determined by X. That is, if two tuples in the relation agree on X, they must also agree on Y [12]. For example, the FD SSN → Student should hold in a relation with columns SSN, Student, and Courses since for any two tuples in the relation, if the SSN attributes match, the Student attributes should also match.

Although functional dependencies are very useful for specifying which data depends on other data, they are limited in that they can express either one-to-one relationships or many-to-one relationships, but not one-to-many or many-to-many relationships. In real world databases, a certain attribute may actually determine a set of values of an attribute as opposed to a single value. Therefore, multivalued dependencies are needed to specify when an attribute determines a set of values [12]. Multivalued dependencies are constraints of the form "X $\twoheadrightarrow$ Y", read as "X multidetermines Y". As an example, the multivalued dependency "Parent $\twoheadrightarrow$ Child" should hold in a relation with columns Parent, Child, and Hobbies (where "Hobbies" are the hobbies of the Parent) since a specific set of children depends on the parent, and is independent of the parent's hobbies.

Another important characteristic of multivalued dependencies is that they allow the designer to express when two things are not directly related [4]. For example, in a relation with columns Parent, Child, and Hobbies, as above, the multivalued dependencies "Parent $\twoheadrightarrow$ Child" and "Parent $\twoheadrightarrow$ Hobbies" express the facts that a set of children depends on a specific parent, independent of the parent's hobbies, and that a set of hobbies depends on a specific parent, independent of the parent's set of children. Thus, these two multivalued dependencies, written "Parent $\twoheadrightarrow$ Child | Hobbies" for short, not only express the relationships between parents and children and parents and hobbies, but they also express the fact that there is no relationship between children and hobbies except the indirect relationship through the parent [4].

The concepts of FDs and MVDs can be directly applied to the process of normalization. In general, FDs are used to "synthesize" 3NF or BCNF relations by using the FDs to determine which closely related attributes should be grouped together [4]. On the other hand, MVDs are used to "decompose" a set of attributes into 4NF by splitting the unrelated attributes into separate relations [4]. For example, using MVDs, the above relation scheme (Parent, Child, Hobbies) would be decomposed into the two smaller schemes (Parent, Child) and (Parent,Hobbies) to separate the unrelated attributes "Child" and "Hobbies". Additionally, since databases often require both FDs and MVDs to express constraints, several researchers have proposed normalization techniques using FDs and MVDs together [2,31,32].

To assist designers with logical design of relational databases, the concepts of dependency theory and normalization have been implemented in several CAD tools. The next section reviews the work that has been done to automate the logical design process. Additional concepts of dependency theory are explained in the section as necessary.

2.3 Computer Aided Database Design Tools

As stated previously, researchers have investigated implementing CAD tools for all phases of database design. Reference [7] contains a comprehensive list of database design tools. The list does not include a detailed review of the capabilities of each tool, therefore, further literature review was required to determine the methods implemented in each tool to design the logical structures of a database. The literature review revealed that several tools have been implemented to support entire database design methodologies [2,6,24]. Additionally, several tools have been developed which focus on the logical design process. Table I contains a representative sample of the recent work which has been accomplished to automate the logical design phase of database design.

The design tools listed in Table I are discussed in more detail as follows:

2.3.1 AFIT Theses. The first tool listed in Table I designs 3NF relation schemes by finding a minimal cover of a set of FDs, and then generating a relation corresponding to each FD in the minimal cover. The resulting relations are automatically in 3NF [29].

A minimal cover is a reduced set of FDs which is equivalent to the original set, but with no redundancies. By equivalent we mean that both sets have the same closure, where the closure of a set F of FDs is the set of all FDs logically implied by F. The closure of F is denoted by F^+.

Tool	Design Algorithms Implemented
1. AFIT Threses [15,19,27]	- Minimal cover of FDs (results in 3NF relations)
2. Scheme Design System (SDS) [17]	- calculate dependency basis - find envelope set of MVDs - find minimal cover of MVDs - find keys and M^- - 4NF decomposition - BCNF decomposition - Nested Normal Form (NNF) decomposition
3. Ceri and Gottlob [8]	- closure of a set of attributes - find minimal cover - determine keys - test for lossless joins - 3NF design - BCNF design
4. Relational Database Design Aid Version 1 (RED1) [6]	- test to determine normal form of database - 3NF design - determine if a particular data dependency logically follows from previously defined dependencies
5. Information Resource Management Aid (IRMA) [10]	- develop data structure charts which are in 3NF
6. Database Design and Evaluation Workbench (DDEW) [24]	- BCNF normalization
7. DATAID [2]	- flow graph approach for logical design
8. Silva and Melkanoff [26]	- Armstrong database instance
9. Data Designer [30]	- 3NF design
10. Information Builder [30]	- 3NF design
11. Design [30]	- Normalization - finds matching keys and combines tables - identifies foreign keys
12. Database Designer's Workbench [9]	- 3NF design

Table 1. Computer Tools for Logical Design of Relational Databases

A set F of dependencies is minimal (referred to as canonical in [16]) if:

1. The right hand side (RHS) of each FD in F is a single attribute.

2. No FD $X \rightarrow A$ in F can be eliminated and still maintain an equivalent set of FDs. That is, if an FD can be removed from F and the closure of $F - \{X \rightarrow A\}$ is equal to the closure of F, then the unnecessary FD must be removed from F.

3. The left hand side (LHS) of each FD in F has been reduced. That is, if an attribute can be removed from the LHS of an FD without changing F^+, then the "extraneous" attribute must be removed.

The concept of minimal cover is central to normalization. As Beeri and Kifer explain "There is a wide consensus in the database community that, for the logical design, one only needs dependencies from some minimal cover" [4, page 138]. They also assert that "It is widely acknowledged that the design process begins by finding a minimal cover of a dependency set" [4, page 142]. One of the main reasons it is important to find a minimal cover, is that since the minimal cover has the same closure as the original set of dependencies specified by the designer, the minimal cover contains all of the same "potential" information as the original set; however, since the redundancies have been removed, the relations generated from a minimal cover should contain less redundancy [8].

An important characteristic of minimal covers is that they are not unique for a given set of dependencies. That is, the minimal cover will vary depending on the order in which dependencies are removed from the original set of dependencies. Thus, alternative designs can be developed by changing the order before generating the minimal cover.

The process implemented in the AFIT theses to find a minimal cover consists of the following steps [15]. First, the RHS of each FD is reduced to a single attribute. Then, if the LHS of two FDs in the new set of FDs determine the same single attribute, and one of the LHSs is a subset of the other, then the FD with the larger set of attributes in the LHS is removed. The third step in the process removes redundant explicit transitive dependencies. That is, if a transitive dependency can be inferred from two dependencies in the set, then the transitive dependency can be removed from the set if it exists explicitly. For example, if $A \rightarrow B$ and $B \rightarrow C$ are two dependencies in the set of FDs, then the transitive dependency $A \rightarrow C$ is implied by these two dependencies. If $A \rightarrow C$ exists explicitly in the set of FDs, it is removed. Jankus claims that the set of FDs which results from these three steps is a minimal cover of the original set, and is in 3NF if each FD is treated as a separate relation [15]. In addition to the steps required to produce the minimal cover, the process has two more steps to reduce redundancy. The fourth step is to combine FDs with the same LHS into a single FD with a RHS which is the union of all the RHSs of the combined FDs. Then, if two FDs contain the same attributes, the fifth step is to eliminate one of the FDs. For example, if the set contains $A \rightarrow B$ and $B \rightarrow A$, then one of the two FDs can be eliminated since both will result in a relation generated with the two attributes A and B. Once the minimal cover is complete, the system then generates a relation for each FD in the cover.

2.3.2 Scheme Design System (SDS). The Scheme Design System implements the following design algorithms to assist the database designer.

2.3.2.1 Dependency Basis. Given a set M of multivalued dependencies, the tool can calculate the dependency basis of a set of attributes X in a universal set U of attributes with respect to M. The dependency basis of an attribute is a set of sets of attributes which can be used to find the set of MVDs of the form $X \twoheadrightarrow Y$ logically implied by M [17]. This algorithm is required to support other algorithms in the SDS such as finding a minimal cover of a set of MVDs.

2.3.2.2 Envelope Set. An envelope set is the set of MVDs which is logically implied by a set D of FDs and MVDs. The generated envelope set of MVDs can be used to decompose relations in the context of both FDs and MVDs [33]. The algorithm implemented in the SDS executes in a time complexity which is polynomial in the size of D [17].

2.3.2.3 Minimal Cover of MVDs. As described above for FDs, the minimal cover of a set of MVDs is a reduced set of MVDs which is equivalent to the original set, but with no redundancies [17]. An MVD X $\twoheadrightarrow$ W in a set M of MVDs is reduced if [17]:

- X is nontrivial. That is, XW does not equal U, or W is not a subset of X.
- The LHS cannot be reduced. That is, there is no subset of X, X', such that $X' \twoheadrightarrow W$ is in the closure of M.
- The RHS cannot be reduced. That is, there is no subset of W, W', such that $X \twoheadrightarrow W'$ is in the closure of M.
- The MVD is nontransferable. That is, there is no subset of X, X', such that $X' \twoheadrightarrow W(X - X')$ in the closure of M.

If every MVD in a set M of MVDs is reduced, and if no proper subset of M is a cover of M, then M is a minimal cover [17].

2.3.2.4 Keys and M^-. For the SDS, M^- is defined as a set of reduced MVDs of M^+, and keys are defined as the set LHS(M^-) [17]. The SDS generates M^- and the set of keys so the keys can be used to decompose a set of attributes into 4NF relation schemes.

2.3.2.5 4NF and BCNF decomposition. The SDS uses a single algorithm to design both 4NF and BCNF relation schemes. If the set of dependencies used for the decomposition process contains both FDs and MVDs, or MVDs only, then the algorithm produces a 4NF decomposition. However, if the set of dependencies contains FDs only then the algorithm produces a BCNF decomposition [17].

2.3.2.6 Nested Normal Form (NNF) decomposition. The NNF decomposition algorithm produces nested relational database schemes, a new research area extending current relational database technology [25].

2.3.3 Ceri and Gottlob. In their normalization tool, Ceri and Gottlob implement several database design algorithms in the Prolog programming language. Unlike the SDS, Ceri and Gottlob do not incorporate the use of multivalued dependencies in their tool, although they indicate that the tool has an "open" design so that new capabilities can be easily added [8]. In order to limit the complexity of their system, they only allow specification of functional dependencies, and thus cannot design normal forms such as 4NF which depend on specification of multivalued dependencies [8]. The major design algorithms they implemented include the following.

They implement an algorithm to find a minimal cover which differs from the one implemented in the AFIT Theses discussed above. Ceri and Gottlob's implementation requires computing the closure of attributes quite often to determine the minimal cover, whereas attribute closure is not computed for the method used in the AFIT theses. The closure of an attribute X with respect to a set of FDs is the set of all attributes functionally determined by X [16]. It is interesting to note that the attribute closure algorithm is used by all of the algorithms in Ceri's and Gottlob's tool [8]. Although algorithms exist to compute attribute closure efficiently (i.e. in a time complexity of $O(N)$ where N is the length of the input [12]), an algorithm which avoids calculating this closure for each LHS of all FDs, and for all subsets of each LHS of all FDs during the left reduction procedure could possibly execute faster for a given set of FDs. Thus, the algorithm implemented in the AFIT theses would appear to be the faster of the two approaches. However, a closer examination of the algorithm implemented in the AFIT theses is required to determine if it accurately generates a minimal cover in all cases. This analysis is shown in Section 4.2.2.

Additionally, as indicated in Table I, Ceri and Gottlob implemented an algorithm to find all the keys of a relation scheme, with keys defined as the attributes of the relation scheme which can uniquely identify each tuple [8]. Identification of keys is important for both designing the relation schemes and also for retrieving data from the database.

The algorithm which tests for lossless joins ensures that a particular decomposition does not result in the loss of any information which was available in the original relations. "The decomposition of one relation R into several relations R_i is called lossless (i.e., possessing the lossless join property) if it is possible to reconstruct R by equijoining the relation R_i over the common attributes..." [8, page 534]. Note that in this context, "equijoining" is equivalent to "natural joining". Ceri and Gottlob indicate that the algorithm they implemented to decompose relations into 3NF does not guarantee losslessness so the losslessness test can fail for some 3NF decompositions [8]. However, other algorithms exist which guarantee the lossless join property [16] and thus this test would not be necessary if it is already incorporated into the decomposition algorithm.

Finally, Ceri's and Gottlob's tool contains algorithms to decompose relations into 3NF and BCNF. The 3NF algorithm implemented was originally defined in [5], and the BCNF algorithm is defined in [28].

2.3.4 Relational Database Design Aid Version 1 (RED1). RED1 is a tool developed at the University of Stockholm for logical design of relational databases. With the tool, a database designer can specify functional and multivalued dependencies, can test whether a database scheme is in 2NF, 3NF, BCNF, or 4NF, can determine if a certain data dependency logically follows from a set of dependencies, or can generate 3NF relation schemes [6]. Like Ceri's and Gottlob's tool, the 3NF generation function in RED1 is based on the algorithm described by Bernstein in [5].

2.3.5 Information Resource Management Aid (IRMA). The IRMA is a tool to assist with logical database design which helps the designer organize data into data structure charts which are in 3NF. The data structure charts are in 3NF (but not 1NF since repeating fields are allowed) because each branch of the chart under a key is a 3NF relation since the concept of functional dependencies is embedded into the logic of data structure chart formation [10]. IRMA does not utilize dependency theory algorithms to normalize the logical structures since the relations are by default in 3NF.

2.3.6 DDEW and DATAID. The Database Design and Evaluation Workbench (DDEW) [24] and the DATAID project [2] are both development efforts to support entire database design methodologies. The systems contain automated tools to assist with all phases of database design.

EMP	DEPT	MGR
Hilbert	Math	Gauss
Pythagoras	Math	Gauss
Turing	Computer Science	von Neumann
Einstein	Physics	Gauss

Figure 1. Example of Armstrong Relation [11, page 20]

For the logical design phase, DDEW supports normalization into BCNF. The DATAID project provides a different approach for logical design. The logical design portion of DATAID translates the conceptual schema, which is based on an extension of the Entity Relationship model, into logical structures by first converting the conceptual schema into a flow graph. In the graph, nodes represent entities and arcs represent relationships between entities. The graph construction process can produce more than one design, so the design which minimizes the number of logical accesses of the operations and the amount of data transferred during I/O operations is selected. Then, the logical structures are generated from the flow graph [1].

2.3.7 Silva and Melkanoff. Reference [26] indicates that Silva and Melkanoff implemented a tool to assist database designers with determining the FDs and MVDs which should hold for a certain set of attributes. To do this, the tool generates an instance of an Armstrong relation which is a relation that obeys precisely every specified dependency but no others [11]. Once an instance of the relation is generated, it is assumed that the user can recognize whether the correct dependencies have been specified or if some are missing or incorrect [26]. The following example from [11] illustrates this very clearly:

Let D be the set of dependencies $\{EMP \rightarrow DEPT, DEPT \rightarrow MGR\}$. Then, the closure of D consists of the FDs in D and others such as $EMP \rightarrow MGR$. The relation in Figure 1 is an Armstrong relation for D because it obeys every FD in the closure of D but no others.

Fagin explains that the designer could examine this relation and determine that "Here is a manager, namely Gauss, who manages two departments. Therefore, the dependencies I inputted must not have implied that no manager can manage two different departments. Since I want this to be a constraint for my database, I'd better input the FD $MGR \rightarrow DEPT$" [11, page 2]. Thus, the generation of an instance of an Armstrong relation for a given set of dependencies could be very useful for determining which dependencies should be specified for a given set of attributes.

Recent information indicates this tool no longer exists [20].

2.3.8 Data Designer, Information Builder, Design. These three design aids are commercially available tools which have been developed to support the normalization process [30]. In addition to normalization, Design also provides algorithms to find relations with matching keys and combine those relations to minimize redundancy. Also, the tool identifies foreign keys which are attributes in a relation r which are not keys of r, but are keys of another relation in the database [13]. This function assists the designer in understanding how the various tables are related.

2.3.9 Database Designer's Workbench. Like DDEW and DATAID, the last tool listed in Table I also supports the entire database design process. The functions provided for the logical design phase include a normalization tool which synthesizes 3NF relations from a set of functional dependencies [9]. Additional logical design tools are planned for future implementation.

2.4 Summary

Designing a database is a very complex set of activities which is time consuming and, if not done properly, can lead to a database which does not perform efficiently or which does not consistently maintain the integrity of the data it is intended to record [15]. Thus, several database design methodologies have been described in the literature to establish a structured approach to the task of database design. Most of the approaches follow clearly defined steps or phases for the design process. And since the process of database design is very complex, these structured design approaches will help to ensure that all important information is considered and ultimately incorporated into the database [15]. Thereby, ensuring the quality and integrity of the database design.

Additionally, since many of the tasks involved in designing a database are time consuming and repetitive, they are good candidates for Computer Aided Design (CAD) tools [19]. Therefore, many researchers have developed computer tools to aid database designers. Although researchers have investigated implementing CAD tools to assist designers in all phases of database design, a significant effort has been applied in the area of logica' design. This is because the process of logical design is well suited for computer assistance because the process can be time consuming, repetitive, and it can be structured into a clearly defined set of steps.

Although the computer tools presented in Table I implement several approaches to logical database design, most of the tools assist with the process of normalization. To accomplish normalization, the tools contain a variety of algorithms to manipulate data dependencies, and to generate the normalized set of relations. The basic set of algorithms needed for this process includes the algorithms for finding a minimal cover of a set of dependencies, generating attribute closure, ensuring decompositions are lossless either by incorporating this check into the decomposition algorithms or by implementing a separate algorithm to check for losslessness, and normalization algorithms for generating the normalized database schemes. Additionally, other algorithms can be very helpful to the designer, such as the one implemented to generate instances of Armstrong relations to help the designer find the dependencies for a database.

Most of the normalization tools presented focus on designing database schemes in one or two normal forms. And, although many researchers have pointed out that decomposition algorithms will generate different designs depending on the minimal cover used as input, none of the tools attempt to generate alternative designs by automatically generating all possible minimal covers.

Additionally, the majority of the tools normalize relations into 3NF or BCNF using only functional dependencies, or into 4NF using only multivalued dependencies. Most tools do not use an approach to normalization which integrates both FDs and MVDs. This limits the types of data dependencies which a designer can specify for a particular set of data.

Thus, although many researchers have developed computer tools to assist with logical design of relational databases, a tool which could automatically generate alternative designs by computing different minimal covers, normalize relations using FDs, MVDs, and both types of dependencies together, and which could help the designer determine the dependencies for a certain set of data would be very useful as a design aid, a research tool, and to students studying database design.

III. Requirements Analysis

3.1 System Objectives

The overall objective of the Dependency Theory Toolbox is to automate algorithms which are used to design logical structures for relational databases, and to provide an interface to the toolbox which helps the user work with the algorithms. The system is intended for use in an academic environment as a teaching aid and research tool rather than for practical application to database design problems. However, the tool could be used to design small relational databases which have a limited number of attributes.

Additionally, a long range objective of the toolbox is to serve as a normalization tool in a suite of stand-alone database design tools developed at AFIT. Therefore, the file formats used by the toolbox must be designed to provide a standard interface so database attributes and constraints can be passed between all tools.

3.2 System User

Since this design tool is mainly for use in an academic environment, the system users will be instructors and students who are studying dependency theory and relational database design. Therefore, the system can be designed for users who are familiar with dependency theory and normalization concepts.

3.3 Functional Requirements

The functional requirements for the toolbox were defined and analyzed using Structured Analysis and Design Technique (SADT). SADT is a methodology for accomplishing functional analysis and system design. In this methodology, SADT diagrams are used to document the system requirements. The requirements analysis was done in a top-down, structured, modular, and hierarchical fashion. That is, the top level requirements were defined first, and then the next lower level in the function hierarchy was defined, and so on, until the lowest level functions were defined. Additionally, the functional requirements were grouped into modules of related activities.

The SADT diagrams which define and document the functional requirements for this toolbox are contained in Appendix B. The diagrams explain "what" the requirements are, and the pages of text which correspond to each diagram explain the requirements in more detail and explain "why" some of the requirements exist.

3.4 Database Design Algorithms Required to Implement Functions

The requirements analysis revealed that the toolbox must contain algorithms to accomplish several functions. Definitions of some of the pertinent terms are presented before the list of algorithms.

- *superkey*: a set of attributes which uniquely identifies each entity (tuple) of a relation. That is, a superkey functionally determines all attributes in the relation.
- *candidate key*: a superkey which has no proper subset which is also a superkey.
- *primary key*: a candidate key which the database designer chooses to use as the primary means to identify each element (entity) in a set of entities.
- *fully dependent*: an attribute is fully dependent on a set of attributes when it is functionally dependent on the entire set of attributes, but not dependent on any subset of the attributes.
- *trivial FD*: an FD, $X \rightarrow Y$ is trivial if $Y \subseteq X$.
- *trivial MVD*: an MVD, $X \rightarrow\rightarrow Y$, is trivial whenever $Y \subseteq X$ or $Y \cup X$ equals all the attributes in the relation being considered.

1. **3NF Designs**. Third Normal Form (3NF) is a normal form in which each relation of a database conforms to the following restrictions [18]. First of all, each relation cannot contain

nonkey attributes which are functionally dependent on part of the primary key for the relation. In other words, each nonkey attribute of each relation must be fully dependent on the primary key. (This requirement causes the relations to be in 2NF). Additionally, no relation can contain a nonkey attribute which is dependent on another nonkey attribute. This last requirement ensures that no nonkey attribute is transitively dependent on the primary key.

2. **BCNF Designs**. Boyce/Codd Normal Form (BCNF) is a stronger normal form than 3NF. BCNF has the same restrictions as named above for 3NF, however, BCNF also requires that the LHS of each nontrivial functional dependency be a superkey of the relation it applies to [16].

3. **4NF Designs**. Fourth Normal Form (4NF) is a normal form which is defined exactly like BCNF except that instead of functional dependencies, the 4NF definition uses multivalued dependencies [16]. That is, the LHS of each nontrivial multivalued dependency must be a superkey of the relation it applies to.

4. **Minimal Cover of a Set of FDs**. A minimal cover of a set of FDs is a reduced set of FDs which is equivalent to the original set, but with no redundancies. See Section 2.3.1 for a detailed definition. This function is required in the toolbox because, as explained in Section 2.3.1, the concept of minimal cover is central to normalization. Minimal covers are important because the cover contains all the same "potential" information as the original set; however, since the redundancies have been removed, the relations generated from a minimal cover should contain less redundancy [8]. This function will be used by the user as a stand-alone function, and by the normalization algorithms.

5. **Minimal Cover of a Set of MVDs**. As described above for FDs, the minimal cover of a set of MVDs is a reduced set of MVDs which is equivalent to the original set, but with no redundancies [17]. See Section 2.3.2.3 for a detailed definition. This function is required in

the toolbox for the same reason as the function described above to find the minimal cover of a set of FDs.

6. **Envelope Set for a Set of FDs and MVDs**. As described in Section 2.3.2.2, an envelope set is the set of MVDs which is logically implied by a set D of FDs and MVDs. The generated envelope set of MVDs can be used to decompose relations in the context of both FDs and MVDs [33]. This function is required in the toolbox because the system is required to design relations in the context of both FDs and MVDs.

7. **Attribute Closure**. As stated in Section 2.3.3, the closure of an attribute X with respect to a set of FDs is the set of all attributes functionally determined by X [16]. This function is required in the toolbox because attribute closure is needed to support other functions required in the toolbox, such as the minimal cover and 3NF design functions.

8. **Membership Algorithm**. This algorithm will determine if an FD is in the closure of a given set of FDs. This function is required in the toolbox to support other functions required in the toolbox, such as the attribute closure and minimal cover functions.

9. **Instance of Armstrong Relation**. This function will generate an instance of an Armstrong relation (see Section 2.3.7) for a given set of dependencies and attributes. This function is required in the toolbox to help the system user determine which dependencies should be specified for a given set of attributes.

10. **Dependency Basis of a Set of Attributes**. The dependency basis of a set of attributes X is the set of sets of attributes logically implied by X with respect to a given set of FDs and MVDs. This function is required in the toolbox to support other toolbox functions such as finding a minimal cover of a set of MVDs.

IV. Design Process

4.1 Required System Modules

The required system modules were derived from the functional requirements defined in the SADT diagrams in Appendix B. The structure charts in Appendix C show the top level modules and their relationships to one another. The toolbox contains many additional lower level modules which support these top level functions. The function of each module is documented in the module headers in the toolbox program listings.

4.2 Algorithm Selection

The requirements analysis revealed that the toolbox must contain algorithms to accomplish the following main functions:

- Envelope Set
- FD/MVD Minimal Cover
- Dependency Basis
- 4NF Decomposition
- BCNF Decomposition
- 3NF Decomposition
- Minimal Cover
- Membership Algorithm
- Attribute Closure
- Armstrong Relation function
- Alternative Logical Designs

For most of the functions, many different algorithms have been published to accomplish each task, therefore, it was necessary to select the one which best suited our objectives. The main selection criteria were:

- time complexity
- implementation time
- availability

Time complexity was an important criteria because we wanted algorithms which would execute in a reasonable amount of time (i.e., polynomial time or faster). Implementation time was important because we needed algorithms which could be implemented within the time limits of this project. Availability was used as a selection criteria because if source code was available for an algorithm, or if detailed pseudo code was available in the literature, there was no reason to redevelop something that was already done as long as the time complexity of the algorithm was acceptable.

The time complexity of an algorithm is a measure of the amount of time required for the algorithm to execute. For this application, we were mainly concerned with the *worst case* running time of the algorithms. The worst case running time, or "order of" an algorithm, can be determined by analyzing the performance of the algorithm with respect to the dimensions of the objects it manipulates. Therefore, an "order of" analysis was done using this approach to compare the running times of algorithms considered for implementation in the toolbox. If the analysis of a specific algorithm was previously done in the literature, the analysis is cited.

The space complexity of the algorithms, that is, the amount of space used by the algorithm, was not a critical factor for most of the algorithms because the space required by the algorithms will not vary significantly. For example, a 3NF design algorithm generates a set of relation schemes, and regardless of how the algorithm generates those schemes, the number of schemes output by different algorithms will not vary significantly. This is also true for other algorithms, such as the minimal cover algorithm, dependency basis algorithm, and other algorithms required in the toolbox. However, space complexity is an important factor for selecting an algorithm to generate alternative designs since this algorithm could potentially generate a large number of database schemes. Therefore, space complexity was considered in the examination of alternative design algorithms.

The following subsections document the algorithm selection process, and the pseudo code for each algorithm selected for implementation.

4.2.1 Envelope Set, FD/MVD Minimal Cover, Dependency Basis, 4NF and BCNF Decomposition. Several of the functions have been implemented in the Scheme Design System [17] which was available at AFIT, and therefore, the algorithms implemented were candidates for the toolbox as long as the time complexity of their execution was polynomial time. The algorithm implemented in the SDS to compute the envelope set has a time complexity which is polynomial in the size of the set of MVDs and FDs used as input [17]. Additionally, the time complexity of the FD/MVD minimal cover algorithm implemented in the SDS is polynomial in the size of the set of dependencies used as input [17].Therefore, these two algorithms were acceptable for implementation in the toolbox.

The SDS also contained a dependency basis algorithm and a single algorithm for generating 4NF or BCNF database designs. These algorithms also execute in polynomial time [17]. Therefore, the algorithms for BCNF decomposition, 4NF decomposition, FD/MVD minimal cover, envelope set, and dependency basis were taken from the SDS and adapted for use in the toolbox. The pseudo code for these algorithms is presented below as it appears in [17].

Dependency Basis

See Section 2.3.2.1 for the definition of dependency basis.

```
INPUT: A set of attributes $U$ and a set of MVDs $M$ on $U$.
OUTPUT: Dependency basis of $X$ with respect to $M$.
begin
```

```
    $S = [U - X]$;
    repeat
        look for dependencies $V \twoheadrightarrow W$ in $M$ and a set $Y$
          in S such that $Y \cap W \neq \emptyset$ but $Y \cap V = \emptyset$;
        replace $Y$ by $Y \cap W$ and $Y - W$ in $S$;
    until no more change can be made to S;
    Output S;
end
```

Envelope Set

See Section 2.3.2.2 for the definition of Envelope Set.

```
INPUT: A set $D$ of MVDs $M$ and FDs $F$.
OUTPUT: The envelope set $E(D)$ of $D$.
begin
    1. Let $F = \{F1, F2, \ldots, F_n\}$, where there are
           no $F_i$ and $F_j (1 \leq i, j \leq n)$ with the same LHS;
    2. if($F = \emptyset$) then return ($M$);
    3. $F_i' = \{X \rightarrow A1, X \rightarrow A2, \ldots, X \rightarrow A_j \mid F_i = (X \rightarrow Y) \in F$
             and $Y = A1A2 \ldots A_j$ where
             $A_k (1 \leq k \leq j)$ is a single attribute$\}$;
       $F' = \bigcup_{i=1}^{n} F_i'$;
       $F'' = \{X \twoheadrightarrow Y \mid X \rightarrow Y$ is in $F'\}$;
    4. $M' = M \cup F''$;
    5. Let LHS($M'$) be $\{X1, X2, \ldots, X_n\}$;
       $M'' = \{X_i \twoheadrightarrow W_p \mid X_i \in LHS(M')$ and $W_p \in DEP_{M'}(X_i)\}$;
       NOTE: $DEP_{M'}(X_i)$ represents the dependency basis of
       $X_i$ with respect to $M'$.
    6. Delete $X \twoheadrightarrow Y$ from $M''$, if:
       a) there is $W \rightarrow Z$ in $F$ such that $W \subseteq X$ and $Y \subseteq Z$; or
       b) there is $S \in DEP_{M'}(X)$ and $Y \subseteq S$,
             and there is $W \rightarrow Z$ in F such that $Y \subseteq Z$ and $W \cap S = \emptyset$
    7. Output $M''$.
end.
```

FD/MVD Minimal Cover

The following algorithm generates the minimal cover of a set of MVDs. (See Section 2.3.2.3 for the definition of an MVD minimal cover.) In order to generate the minimal cover of a set of dependencies which contains both FDs and MVDs, the envelope set of the dependencies must be computed as shown in the above algorithm, and then provided as input to this algorithm.

INPUT: A set of attributes U and set of MVDs M on U.

```
OUTPUT: The minimal cover of set $M$.
begin
1. {eliminate reducible attributes}
   For (each $X \in LHS(M)$) do
   begin
   1.1. {eliminate trivial MVDs}
        For (each $X \twoheadrightarrow V \in M$) do
        if ($V \subseteq X$ or $XV = U$) then delete $X \twoheadrightarrow V$ from $M$;
   1.2. {obtain right reduced MVDs}
        Calculate $DEP(X)$;
        Replace $X \twoheadrightarrow V$ by $X \twoheadrightarrow V_1 \mid V_2 \mid \dots \mid V_n$ where
        $V_1 V_2 \dots V_n = V$ and $V_i (1 \leq i \leq n) \in DEP(X)$;
   1.3. {obtain left-reduced and transferable MVDs}
        For (each $A \in X$) do
        begin
            calculate $DEP(X - A)$;
            if (there is $V \in DEP(X - A)$ and $V_1 \subseteq VA$) then
                  replace $X \twoheadrightarrow V$ by $(X - A) \twoheadrightarrow V_1$;
        end
   end
2. {delete redundant MVDs}
   For (each $X \twoheadrightarrow V \in M$) do
   begin
        $M' = M - (X \twoheadrightarrow V)$;
        if ($V \in DEP_{M'}(X)$) then
            delete $X \twoheadrightarrow V$ from $M$;
   end
3. Output $M$;
end
```

4NF and BCNF Decomposition

Only one algorithm is needed to generate 4NF and BCNF decompositions. The schemes generated by the algorithm are in 4NF if the input set of dependencies includes MVDs, and they are in BCNF if the input set of dependencies includes FDs only. (See Section 3.4 for the definitions of 4NF and BCNF.) The algorithm is preceded by pertinent definitions from [17].

- D: an input set of FDs and/or MVDs.
- M: minimum cover of the envelope set E(D).
- M^-: $\{X \twoheadrightarrow W | X \twoheadrightarrow W$ is a reduced MVD in $M^+\}$.
- ***keys***: LHS(M^-).

- M': set of MVDs which results from randomly selecting one MVD
- $X \twoheadrightarrow W$ from M^- for each $X \in LHS(M^-)$.
- M^*: $M^* = M^- - M'$. M^* is a 4NF covering of M [33].
- *sp-ordering*: a sequence of elements $X_1, X_2, \ldots, X_n$ is a *sp-ordering* if:

  (1) $X_i \subseteq X_j$ implies $1 \leq i < j \leq n$, and

  (2) if D logically implies $X_j \rightarrow X_i$ but D does not

  logically imply $X_i \rightarrow X_j$,

  then $1 \leq i \leq j \leq n$.

```
INPUT: A universal set of attributes $U$ and $M^*$;
OUTPUT: 4NF decompositions over $U$;
begin
  1. Let $LHS(M^*) = \{X_1, X_2, \ldots, X_n\}$ be in sp-ordering;
  2. $R = \{U\}$;
  3. For i = 1 to n do
     begin
        if there is a $U1$ in $R$ such that $U1$ is
           decomposable with respect to $X_i \twoheadrightarrow Y$ in $M^*$
        then replace $U1$ by $U1 \cap X_i Y$ and $U1 - (Y - X_i)$;
     end
  4. Output R;
end
```

4.2.2 3NF Decomposition, Minimal Cover, Membership Algorithm, Attribute Closure. Another database design tool which was available at AFIT (see reference [15]) contains a 3NF algorithm which was considered for implementation in the toolbox. However, analysis of the algorithm revealed that it did not compute closures when performing left reductions and when eliminating redundant attributes. This could lead to generation of schemes which are not in 3NF. Specifically, as described in Section 2.3.1, in order for a set of FDs to be minimal, the LHS of each FD must be reduced. That is, if an attribute can be removed from the LHS of an FD without changing F^+, then the extraneous attribute must be removed [16]. The algorithm in [15] does not remove attributes and then compute the closure, it just looks to see if a subset of the LHS of an FD determines the same RHS and is an FD in the given set. If so, the FD with the larger LHS is removed. The problem with this is that FDs may be implied by the LHS which are not present in the set of FDs, and so extraneous attributes may be left in the LHS if the implied FD does not exist explicitly.

For example, if the given set of FDs, F, is:

$$AB \rightarrow D$$

$$B \rightarrow C$$

$$C \rightarrow D$$

AB → D should be reduced to B → D since B → D is implied by B → C and C → D through transitivity. However, the algorithm in [15] will not reduce AB → D because B → D does not exist explicitly in F, and the algorithm does not compute the closure to see if B → D is in F^+. The relation created for AB → D (i.e., ABD) is not in 3NF since a subset of the LHS of AB → D also determines D. That is, all nonkey attributes of ABD (i.e., D) are not fully dependent on the key (i.e., AB). (see Section 3.4 for definition of 3NF). Since a 3NF relation cannot have any attribute that is functionally dependent on only part of the key (such as B → D in the example), ABD is not in 3NF, and thus the database design is not in 3NF. Thus, it is necessary to compute attribute closure for the normalization process since implied dependencies can cause relation schemes to not be in a certain normal form.

Another problem with the 3NF algorithm implemented in [15] is no steps are taken to ensure all attributes in the universal set of attributes are represented in a relation. For example, if the given set of FDs, F, is:

$$AB \rightarrow D$$

$$B \rightarrow C$$

$$C \rightarrow D$$

$$B \rightarrow D$$

AB → D will be eliminated since B → D is explicitly represented in F. However, now no FD in F contains attribute A, so when relations are generated for each FD, no relation will contain attribute A. This problem also occurs if the universal set of attributes contains attributes which are not contained in any of the FDs which hold on those attributes. For example, if the universal set of attributes is A B C D E and the set of FDs which hold on the attributes is:

$$AB \rightarrow D$$

$$B \rightarrow C$$

$$C \rightarrow D$$

Then, since no FD contains E, no relation generated will contain E.

Due to these problems, other 3NF algorithms were reviewed for implementation. Other 3NF algorithms examined [5,8,16,3] all started with finding a minimal cover as the first step, therefore, it was essential to find an efficient algorithm for computing a minimal cover.

A linear time, $O(n)$, membership algorithm which can be used to produce the minimal cover is presented in [3]. The purpose of the membership algorithm is to determine if an FD is in the closure of a set of FDs. Thus, the membership algorithm can be used to left reduce FDs, and to remove redundant FDs from the set of FDs in the following manner.

The linear time membership algorithm can be used to left reduce FDs, that is, to remove extraneous attributes, B, from the LHS of an FD (say, LHS → A) by testing to see if (LHS - $\{B\}$) → A is in F^+. A simple procedure for accomplishing left reduction follows [3]:

Left Reduction of FDs:

```
$X'$ = LHS of an FD
do for each $B \in$ LHS of the FD;
   if ($X' - \{B\} \rightarrow A$ is in $F^+$)
      then $X' = X' - \{B\}$
end
```

The time complexity of the left reduction procedure is derived as follows in [3]. The set of FDs, F, contains FDs $\{f_1, f_2, \ldots, f_n\}$. Each FD f has attributes on its left hand side and right hand side from the universal set of attributes $\{A_1, A_2, \ldots, A_m\}$. If the attributes are represented by integers $\{1, 2, \ldots, m\}$, then F can be represented as a string of pairs of integers representing the LHS and RHS of each FD. The length of this representation of F is denoted by $|F|$. Additionally, since each attribute in an FD f appears in at least one FD (i.e., at least in f), then $|f| \leq |F|$, where $|f|$ denotes the number of attributes in f [3].

In the left reduction procedure, each extraneous attribute is eliminated in time $O(|F|)$ since the membership algorithm can compute the closure of a set of attributes in time $O(|F|)$. (The analysis of the membership algorithm is given below). Then, since each attribute on the LHS of each FD must be checked, the entire reduction procedure for the set of FDs, F, takes time $O(|F|^2)$ [3].

The membership algorithm can also be used to eliminate redundant FDs by testing to see if an FD, f, is in the closure of F - {f} [3]. The following procedure can be used to remove redundant FDs from F [3]:

Delete Redundant FDs:

```
begin
   $G = F$;
   do for each $f \in F$;
       if $f \in (G - \{f\})^+$ then $G = G - \{f\}$
end
```

The time complexity of the above algorithm is $O(n|F|)$, where n is the number of FDs in F [3]. This follows from the fact that the membership algorithm must be executed once for each FD in F.

The linear time membership algorithm is shown in the following pseudo code as it is presented in [3]:

Linear Time Membership Algorithm for FDs:

INPUT: A set F of n FDs on attributes $\{A_1, \ldots, A_m\}$ and
an FD f: $X \rightarrow A$.
OUTPUT: "YES" if $f \in F^+$; "NO" if $f \in F^+$.

DATA STRUCTURES:

1. Attributes are represented by integers between 1 and m.
2. FDs in F are represented by integers between 1 and n.
3. LS[1:n], RS[1:n] are arrays of sets containing the attributes on the left and right sides of each FD.
4. DEPEND is a set of attributes found to be functionally dependent on X so far.
5. NEWDEPEND is a subset of DEPEND that has not yet been examined.
6. COUNTER[1:n] is an array containing the number of attributes on the left side of each FD that have not yet been found to be in DEPEND.
7. ATTRLIST[1:m] is an array of lists of FDs specifying for each attribute the FDs with that attribute on their left sides.

ALGORITHM:

```
begin
    INITIALIZE: do i = 1 to $m$;
        ATTRLIST[$m$] = 0
    end
    do i = 1 to $n$;
        COUNTER[$i$] = 0;
        do for each $j \in LS[i]$;
            ATTRLIST[$j$] = ATTRLIST[$j$] $\cup \{i\}$;
            COUNTER[$i$] = COUNTER[$i$] + 1;
        end
    end
    DEPEND = X;
    NEWDEPEND = DEPEND;
FIND_NEW_ATTR:
    do while (NEWDEPEND $\neq \emptyset$)
        select NEXT_TO_CHECK from NEWDEPEND;
        NEWDEPEND = NEWDEPEND – {NEXT_TO_CHECK};
    CHECK_FDS:
        do for each $i \in$ ATTRLIST(NEXT_TO_CHECK)
```

```
          COUNTER[$i$] = COUNTER[$i$] − 1;
          if (COUNTER[$i$] = 0)
          then do for each $j \in RS[i]$;
             if ($j \in$ DEPEND)
             then begin
                DEPEND = DEPEND $\cup \{j\}$;
                NEWDEPEND = NEWDEPEND $\cup \{j\}$
             end
          end
       end CHECK_FDS
    end FIND_NEW_ATTRS
PRINT:
    if $A \in$ DEPEND
       then print "YES"
       else print "NO"
end
```

The time complexity of the above algorithm is derived as follows [3]. The INITIALIZE routine has a time complexity of $0(|F|)$ since it basically consists of stepping through each FD in F and performing a constant set of operations for each attribute in the LHS of the FDs. The complexity of INITIALIZE is added to the complexity of FIND_NEW_ATTR which is $O(|F|)$ since "for each attribute in NEWDEPEND, the FIND_NEW_ATTR loop follows a constant number of steps for each occurrence of that attribute on the left side of an FD in F. Similarly, each right side of an FD in F is visited at most once in FIND_NEW_ATTR"[3, page 47]. Thus, the time complexity of FIND_NEW_ATTR is $O(|F|)$ so the time complexity of the entire algorithm is $O(|F|)$ [3].

The above algorithms for left reduction, deleting redundant FDs, and computing membership met all three of the algorithm selection criteria. That is, they all had a reasonable time complexity, reasonable estimated implementation time, and the pseudo code was available in the literature. Therefore, they were chosen for implementation in the toolbox to support the minimal cover function, the 3NF decomposition function, the membership algorithm function, and the attribute closure function.

Now, with an efficient way to find out if FDs are in the closure of a set of FDs, the minimal cover algorithm can be implemented in the toolbox as follows:

FD Minimal Cover:

```
INPUT: a set $F$ of FDs.
OUTPUT: a minimal cover of $F$.

begin
   right reduce FDs (i.e. ensure each FD has only one attribute on its RHS);
   left reduce FDs;
   delete redundant FDs;
end
```

The time complexity of the minimal cover algorithm is derived as follows. The right reduction procedure has a time complexity of $O(|F|)$ since it consists of stepping through each FD in F and performing a constant number of operations based on the number of attributes in the RHS of each FD. Then, as stated above, the time complexity of the left reduction is $O(|F|^2)$, and the time complexity of the deletion of redundant FDs is $O(n|F|)$ where n is the number of FDs in F. We know that $n < |F|$ because each FD must contain at least two distinct attributes since the minimal cover algorithm removes trivial dependencies such as $A \rightarrow A$. Therefore, $|F|$ will always contain at least two attributes for each one FD, and so $|F|$ will always be larger than n. Thus, the time complexity of the entire algorithm is $O(|F|^2)$.

The 3NF decomposition algorithm selected for the toolbox is adapted from an algorithm presented in [16]. However, the following two steps were added to minimize redundancy. First, schemes with identical keys were merged into single schemes. And second, when duplicate schemes occur, one of them is eliminated. (Note: duplicate schemes will occur when two FDs in F have equal inverted LHS's and RHS's, such as $A \rightarrow B$ and $B \rightarrow A$. They will both result in scheme AB so one of the AB schemes should be eliminated.) The pseudo code for the 3NF algorithm follows:

3NF Decomposition Algorithm:

```
INPUT:    a universal set of attributes (universal relation scheme)
          and a set $F$ of FDs which hold on the universal attributes.
OUTPUT: a set of relation schemes $\mathcal{R}$ in 3NF.

$F_c$ = a minimal cover of $F$
$\mathcal{R} = 0$;
```

```
for each FD $X \rightarrow Y$ in $F_c$ do
    begin
        $\mathcal{R} = \mathcal{R} \cup \{XY\}$
    end

Combine schemes in $\mathcal{R}$ with identical candidate keys;

if none of the schemes in $\mathcal{R}$ contains a candidate key for R
    then begin
        $\mathcal{R} = \mathcal{R} \cup$ {any candidate key for R};
    end
if two schemes are equal
    then delete one of the schemes

return($\mathcal{R}$)
```

The time complexity of this 3NF decomposition algorithm is derived as follows. The minimal cover procedure has a complexity of $O(|F|^2)$, as described above. Then, building a scheme for each FD is of $O(n)$, where n is again the number of FDs in F, since this function merely scans the set of FDs F and performs a constant number of operations for each FD to create the schemes. Merging schemes with identical keys has a time complexity of $O(n^2)$ since the key of each scheme (one scheme per FD) must be compared to the key of each of the other schemes and then a constant number of operations are done when two schemes' keys are identical. Next, ensuring that at least one scheme contains a candidate key requires two steps. First, the relations must be scanned to see if one of them already contains a candidate key. This is done by using the membership algorithm to see if the key for the particular scheme determines all of the attributes of the universal relation. Thus, since the membership algorithm's complexity is $O(|F|)$, and the worst case number of schemes is equal to n (that is, if no identical keys were merged), then the time complexity of searching the relations for one that contains a candidate key is $O(n|F|)$. Second, if no relation contains a candidate key, one must be found and put into a new relation. A candidate key can be found by creating a trivial FD with both the LHS and the RHS equal to the universal set of attributes, then left reducing the trivial FD. This leaves a LHS which determines the entire set of attributes but does not contain any extraneous attributes. Thus, the LHS is a candidate key. This process has a time complexity of $O(|F|^2)$ since it basically involves executing the left reduction algorithm one time. The last function, removing duplicate schemes, is of $O(n^2)$ since each scheme must be compared to every other scheme. Since we know that $n < |F|$, as described above, then $n < |F|$, $n^2 < |F|^2$, and $n^2 < n|F|$. Thus, combining the time complexity of each portion of the algorithm results in a time complexity of $O(|F|^2)$ for the entire 3NF decomposition algorithm.

Another 3NF algorithm is presented in [3], however, since its execution time is also $O(|F|^2)$ [3], it offered no advantages over the algorithm described above, and thus the above algorithm was selected for implementation in the toolbox.

The membership algorithm function and the attribute closure function were both implemented using the linear time membership algorithm described above. The membership algorithm function could of course use the algorithm directly. The attribute closure function could be implemented using the membership algorithm because when the algorithm terminates, the variable DEPEND contains the closure of the LHS of the FD passed to it. Thus, to compute the closure of a set of attributes X, the attribute closure function must simply pass the membership algorithm a trivial FD X → X. When the membership algorithm terminates, DEPEND will contain the closure of X with respect to the given set of FDs. Since both functions basically consist of executing the membership algorithm one time, they are both $O(|F|)$ functions.

4.2.3 Instance of an Armstrong Relation. The function which generates instances of Armstrong relations contains several main algorithms. A review of the literature on Armstrong relations revealed that only one source had published a description of the algorithms in enough detail to develop a working function for the toolbox. The article [26] indicates that a tool which could generate instances of Armstrong relations had been developed at UCLA, however, more recent information indicates the tool no longer exists [20]. Since only one source of the algorithms was available, the algorithm selection criteria did not need to be applied, and an analysis of the time complexity was not necessary for comparison purposes. A description of the required functions and the pseudo code for the required algorithms follows. The description and the pseudo code are drawn directly from [26], and thus the presentation represents a summary of the work done by Silva and Melkanoff.

The general steps required to generate an instance of an Armstrong relation are shown in the high level pseudo code below. The code defines the driver for the entire function. The pseudo code for each function listed in the driver is presented in subsequent sections.

Armstrong Instance Driver:

```
INPUT:    a universal set of attributes (universal relation)
          and a set of FDs and/or MVDs which hold on the universal
          relation.
OUTPUT:   an instance of an Armstrong relation for the given input.

begin
   1. Set $L = 0$, where $L$ is a counter used by the decomposition
      algorithm in Step 2. A unique value of $L$ is associated with
      each relation generated, and the same value of $L$ is associated
      with the set of FDs which hold on each relation.

   2. Decompose the universal relation into a dependency preserving,
      lossless join decomposition.

   3. Rearrange the relations generated by the decomposition algorithm
      so they are in a form which can easily represent the
      relationships among each attribute.

   4. Compute the minimum domain size of each attribute required to
      represent the FDs and MVDs.

   5. Build a unary relation for each attribute with the minimum number
      of values in each relation.

   6. Generate an instance for each relation in the database
      scheme with each instance showing the appropriate relationships
      between its attributes.

   7. Compute the natural join of all the relation schemes to
      create an instance of the universal relation. This instance is
      an instance of an Armstrong relation.

   8. Display the instance of the Armstrong relation to the user.
end
```

4.2.3.1 Integrated FD/MVD Decomposition. As shown in the steps above, in order to generate an instance of an Armstrong relation, the universal relation must first be decomposed into a dependency preserving, lossless join relation scheme. The decomposition algorithm used by Silva and Melkanoff in [26] is defined by Melkanoff and Zaniolo in [21]. This same algorithm was implemented in the toolbox because the remaining steps needed to generate the instance depend on the specific cases which result from the Zaniolo/Melkanoff algorithm (explained below). Thus, if a different decomposition algorithm was used, say the 4NF/BCNF algorithm already implemented in the toolbox, the resulting schemes would not represent the cases which the other algorithms in the Armstrong relation function require, and thus all new algorithms would have to be written.

The pseudo code for the decomposition algorithm from [21] is shown below, and is preceded by pertinent definitions.

- *elementary FD*: an FD of R, $X \rightarrow A$, is called elementary if A is not an element of X, and R contains no $X' \rightarrow A$ where $X' \subset X$.

- *elementary MVD*: an MVD of R, $X \twoheadrightarrow Y$ is called elementary if Y is non-empty and disjoint from X, and R does not contain another MVD, say $X' \twoheadrightarrow Y'$, where $X' \subseteq X$ and $Y' \subseteq Y$. Note: according to the above definition, a trivial MVD can be elementary [34]. This is an important point for the decomposition algorithm since it determines the elementary MVDs which apply to specific schemes.

- *multiple elementary MVDs*: an elementary MVD of R is called multiple if R contains other elementary MVDs with the same LHS; otherwise it is called single.

The following variables must be defined for use in the decomposition algorithm:

- W: the set of attributes to be decomposed.

- $R(W)$: refers to the relation scheme consisting of the set of attributes W.

- F: the set of elementary FDs of $R(W)$, that is, the elementary FDs which hold on scheme $R(W)$.
- F_s: the set of elementary FDs in F having W as scope, where the scope of an FD, $X \rightarrow Y$, is $\{X \cup Y\}$. Thus, if an FD, $X \rightarrow Y$, has scope W, $\{X \cup Y\} = W$.
- $F1$: the set of elementary FDs of $R[W1]$, where $W1 \subset W$. The decomposition algorithm computes $W1$ when decomposing W.
- $F2$: the set of elementary FDs of $R[W2]$, where $W2 \subset W$, and is also computed by the decomposition algorithm.
- G_m: the set of multiple elementary MVDs of $R[W]$.
- $G11$: the set of elementary MVDs of $R[W1]$ which have right side disjoint from $W2$.
- $G22$: the set of elementary MVDs of $R[W2]$ which have right side disjoint from $W1$.
- G_F: denotes the set of MVD counterparts of $F1 \cup F2$ (the MVD counterpart of a FD, $X \rightarrow Y$, is the MVD: $X \rightarrow\rightarrow Y$.)
- F^+: denotes the set of FDs implied by the set of FDs F.
- G^+: denotes the set of MVDs implied by the set of MVDs G.
- $ACOVER$: the set of atomic relations generated by the decomposition algorithm, where atomic means the relations cannot be decomposed any further by this algorithm.
- $ZCOVER$: the set of all FDs associated with the relations.
- L: a counter which must be initialized to 0 in the calling program, prior to calling this decomposition algorithm.

Decomposition Algorithm

INPUT: a set of attributes W, and a set of FDs and/or MVDs which
hold on those attributes.
OUTPUT: $ACOVER$ and $ZCOVER$.

```
procedure DECOMPOSE(W)
    begin
        STEP1: DETERMINE(F,$G_M$);
        STEP2: FLAG = false;
                  for each $X \rightarrow A \in F$ do
                  if $X \cup \{A\} = W$ then
                     begin
                         FLAG = true;
                         $ZCOVER = ZCOVER \cup \{L : X \rightarrow A\}$;
                         $F = F - \{X \rightarrow A\}$
                     end STEP2;
                  if $G_m = 0$ then
                     STEP3:
                         begin
                             $ACOVER = ACOVER \cup \{L : W\}$;
                             $L = L + 1$;
                         end STEP3;
                  else
                     STEP4:
                         begin
                             NOTFOUND = true;
                                    for each $X \twoheadrightarrow Y \in G_m$ while NOTFOUND do
                                        begin
                                            $W1 = X \cup Y$;
                                            $W2 = W - Y$;
                                            COMPUTE($F1, F2, G_F, G11, G22$);
                                            if (F - $F_s$) $\subseteq (F1 \cup F2)^+$ and
                                              $G_m \subseteq (G_F \cup G11 \cup G22)^+$ then
                                            begin
                                              if FLAG then $L = L + 1$;
                                              DECOMPOSE($W1$)
                                              DECOMPOSE($W2$)
                                              NOTFOUND = false;
                                            end
                                        end;
                                    if NOTFOUND then REPORTFAILURE
                         end STEP4
     end DECOMPOSE;
```

The decomposition algorithm will call the REPORTFAILURE routine if it cannot find a suitable decomposition which will preserve all dependencies. Otherwise, it will generate a set of relations (referred to as the ACOVER), and the set of FDs which hold on those relations (referred to as the ZCOVER). The ZCOVER is a minimal cover of the FDs which were given to hold on the universal set of attributes.

The DECOMPOSE algorithm generates relations which can have the following relationships among their attributes [26].

- *one-to-one*: every different X value corresponds to a different Y value, that is, $X \rightarrow Y$ and $Y \rightarrow X$.
- *many-to-one*: one or more X values may correspond to the same Y value, but no X value corresponds to more than one Y value, that is, $X \rightarrow Y$ and Y does not determine X.
- *many-to-many*: X does not determine Y and Y does not determine X. That is, one or more X values may correspond to the same Y values and one or more Y values may correspond to the same X value.
- *combined relationships*: a combined relationship exists among three combinations (sets of attributes) X, Y, and Z when $\{X, Y\} \rightarrow Z$ and one of the following cases occurs:

    1. $Z \twoheadrightarrow X$ and $Z \twoheadrightarrow Y$
    2. $Z \rightarrow X$ and $Z \twoheadrightarrow Y$
    3. $Z \twoheadrightarrow X$ and $Z \rightarrow Y$

In order to generate instances of relations which have the above types of relationships, the relations and their associated FDs must be rearranged as follows.

4.2.3.2 Rearrangement Procedures. As shown in the driver for the Armstrong instance, after the decomposition process, the relations must be rearranged to facilitate generation of the instance. Each rearrangement procedure is described below as it is presented in [26]. Additionally, each procedure is represented by pseudo code following the description.

Rearrangements Due to One-to-One Relationships

"If there is a one-to-one relationship between two combinations X and Y that is $X \rightarrow Y$ and $Y \rightarrow X$, and $|X| > 1$ or $|Y| > 1$, then there may be an FD $X \rightarrow Y$ or $Y \rightarrow X$ in the *ZCOVER* without an associated atomic component, and $|X|$ or $|Y|$ additional FDs in the *ZCOVER* with associated atomic components. The *ACOVER* and *ZCOVER* must be rearranged so that only one atomic component appears with two FDs with equal inverted left and right hand sides."[26, page 120]

The pseudo code for this operation follows.

```
begin
   If there are two FDs, $X \rightarrow Y$ and $Y \rightarrow X$, in the
      $ZCOVER$ and $[|X| > 1$ or $|Y| > 1]$.
      then
         1. Search the $ZCOVER$ for an FD, $X \rightarrow Y$ or $Y \rightarrow X$,
            which has no associated relation in the $ACOVER$.
         2. Then search the $ZCOVER$ for $|X|$ or $|Y|$ additional FDs
            which have relations in the $ACOVER$.
            (NOTE: this will occur if the RHS of one of the FDs
            had more than one attribute since the FD will be right
            reduced into $|RHS|$ FDs).
         3. Merge all schemes found in Step 2 into a single scheme
            by computing the union of the attributes in the schemes,
            and add the single scheme to the $ACOVER$.
         4. Associate the two FDs, $X \rightarrow Y$ and $Y \rightarrow X$
            in the $ZCOVER$, with the single scheme just added
            to the $ACOVER$.
end
```

Rearrangements Due to Many-to-One Relationships

"If there is a many-to-one relationship between two combinations X and Y, that is, $X \rightarrow Y$ and $|Y| > 1$, then there may be $|Y|$ FDs in the *ZCOVER*. The *ACOVER* and *ZCOVER* must be rearranged so that only one atomic component with one associated FD appears."[26, page 120]

"If $X \rightarrow Y$ is an FD associated with atomic component $R[X,Y]$, $Z \rightarrow W$ is another FD associated with atomic component $R[Z,W]$ and X contains Z, the *ACOVER* and *ZCOVER* must be rearranged so that only one atomic component $R[X,(Y \cup W)]$ appears with the FDs $X \rightarrow \{Y \cup W\}$ and $Z \rightarrow W$."[26, page 120]

These two steps are accomplished by the following pseudo code.

```
begin
   If there is an FD, $X \rightarrow Y$, in the $ZCOVER$, and $|Y| > 1$,
      then merge all schemes which have $X$ as their key (that is,
      merge equivalent keys).
   For each FD $X \rightarrow Y$ in the $ZCOVER$ associated with $R[X,Y]$ in the $ACOVER$
      begin
         For each FD $Z \rightarrow W$ in the $ZCOVER$ associated with $R[Z,W]$ in the $ACOVER$
            begin
               If $Z \subset X$
                  then
                     add $R[X,(Y \cup W)]$ to the $ACOVER$
                        and the associated FDs $X \rightarrow \{Y \cup W\}$
                        and $Z \rightarrow W$ to the $ZCOVER$;
                     delete $R[X,Y]$ and $R[Z,W]$ from the $ACOVER$
                        and the associated FDs $X \rightarrow Y$ and $Z \rightarrow W$
                        from the $ZCOVER$;
            end
      end
end
```

Rearrangements Due to Combined Relationships

"If there is a combined relationship among combinations X, Y, and Z, then there is an FD $\{X,Y\} \rightarrow Z$ in the $ZCOVER$ with no associated atomic component and 2 atomic components $R[X,Z]$ and $R[Y,Z]$ which belong to one of these three cases:

1. $R[X,Z]$ and $R[Y,Z]$ have no FDs.

2. $R[X,Z]$ has no FD and $R[Y,Z]$ has the FD $Z \rightarrow Y$.

3. $R[X,Z]$ has the FD $Z \rightarrow X$ and $R[Y,Z]$ has no FDs.

The $ACOVER$ and $ZCOVER$ must be rearranged so that only one atomic component appears associated with the FD $\{X,Y\} \rightarrow Z$. The MVD $Z \twoheadrightarrow X$ is added to the $ZCOVER$ if there is no FD $Z \rightarrow X$, while the MVD $Z \twoheadrightarrow Y$ is added to the $ZCOVER$ if there is no FD $Z \rightarrow Y$."[26, page 120]

The pseudo code for this procedure follows.

```
begin
   for each FD $\{X,Y\} \rightarrow Z$ in the $ZCOVER$
      begin
```

```
        if there is no associated relation in the $ACOVER$
          then
             add $R[X,Y,Z]$ to the $ACOVER$ associated with
               FD $\{X,Y\} \rightarrow Z$ in the $ZCOVER$;
             if $R[X,Z]$ and $R[Y,Z]$ are relations in the
               $ACOVER$ with no associated FDs
               then
                  add MVDs $Z \twoheadrightarrow X$ and $Z \twoheadrightarrow Y$ to the
                    $ZCOVER$ associated with $R[X,Y,Z]$
                    in the $ACOVER$;
             else if $R[X,Z]$ has no FDs and $R[Y,Z]$ has the FD $Z \rightarrow Y$
               then
                  add MVD $Z \twoheadrightarrow X$ to the $ZCOVER$ associated
                    with $R[X,Y,Z]$ in the $ACOVER$;
             else if $R[X,Z]$ has the FD $Z \rightarrow X$ and
                  $R[Y,Z]$ has no FDs
               then
                  add MVD $Z \twoheadrightarrow Y$ to the $ZCOVER$ associated
                    with $R[X,Y,Z]$ in the $ACOVER$;
             delete $R[X,Z]$ and $R[Y,Z]$ from the $ACOVER$;
     end
end
```

4.2.3.3 Computing Minimum Domain Sizes. Once the ACOVER and ZCOVER are rearranged, instances of each relation can be generated. However, before generating the instances, the minimum domain sizes for each attribute must be computed to ensure the Armstrong relation will be the minimum size relation which can represent all the relationships among the attributes.

The following algorithm is used to compute the minimum domain size for each attribute. As defined in [26] for the algorithm, "F is a boolean variable which indicates if the computation is to be done again. For each attribute A in relation R(V) there is a variable CT(A) which will contain the minimum domain size when the algorithm is over. When processing an atomic component R[X,Y], X and Y are respectively the left and right side of an FD X → Y associated with R[X,Y]. SX and SY are respectively the cardinalities of relations R(X) and S(Y) corresponding to the left and right sides of the FD with many-to-many relationships among their attributes. LX and LY are respectively the indices of the last attributes (from left to right) of X and Y."[26, page 128]

The following algorithm is presented as it appears in [26, page 128]:

```
1. [Initialization] F $\leftarrow$ TRUE. For each attribute A in V,
   $CT(A) \leftarrow 2$.
2. [Repeat or terminate] If F = TRUE then F $\leftarrow$ FALSE and execute step 3, otherwise terminate the algorithm.
3. [Process atomic component] For each atomic component R[X,Y] determine which relationship exists among its attributes:
   If there are two FDs in the ZCOVER with equal inverted
      left- and right-hand sides, then there is a one-to-one relationship; execute step 4 and step 6.
   If there is an MVD in the ZCOVER, then there is a combined
      relationship; execute step 4 and step 7.
   If there is one or more FDs with different alternated
      sides and no MVD in the ZCOVER, then there is a many-to-one relationship; execute step 4 and step 5.
   If there is no FD in the ZCOVER then there is a many-to-many
      relationship; so do nothing.
   Return to step 2 after all atomic components have been processed.
4. [Compute SX, SY, LX, and LY] Let $X \rightarrow Y$ be an FD in the ZCOVER associated with the atomic component R[X,Y].
      SX $\leftarrow$ 1. For each attribute A in X, SX $\leftarrow$ SX+CT(A)$-$1.
      SY $\leftarrow$ 1. For each attribute B in Y, SY $\leftarrow$ SY+CT(B)$-$1.
      LX $\leftarrow$ last attribute in X.
      LY $\leftarrow$ last attribute in Y.
5. [Many-to-one relationship]
      For each attribute A in X, if CT(A) $\leq$ SY then
```

```
        $CT(A) \leftarrow SY+1$, $F \leftarrow TRUE$.
6. [One-to-one relationship]
    For each attribute A in X if $CT(A) \leq |Y|$ then
        $CT(A) \leftarrow |Y| + 1$, $F \leftarrow TRUE$.
    For each attribute B in Y if $CT(B) \leq |X|$ then
        $CT(B) \leftarrow |X| + 1$, $F \leftarrow TRUE$.
    If $SX<SY$ then $CT(LX) \leftarrow CT(LX)+SY-SX$, $F \leftarrow TRUE$.
    If $SY<SX$ then $CT(LY) \leftarrow CT(LY)+SX-SY$, $F \leftarrow TRUE$.
7. [Combined relationship]
    For each attribute A of the atomic component R[X,Y], if
        $CT(A)<3$ then $CT(A) \leftarrow 3$, $F \leftarrow TRUE$.
    If $SY<SX-2$ then $CT(LY) \leftarrow CT(LY)+SX-SY-2$, $F \leftarrow TRUE$.
    If $SX<SY+2$ then $CT(LX) \leftarrow CT(LX)+2-SX$, $F \leftarrow TRUE$.
```

4.2.3.4 Building Unary Relations. Once the minimum domain sizes are computed, a unary relation is built for each attribute. The relation contains the minimum number of values determined by the above algorithm. Then, instances for the relations which were generated by the decomposition algorithm are built from the unary relations as described in the following section.

4.2.3.5 Generating Instances. Instances of each relation are built based on the type of relationships which exist among its attributes. The procedures for constructing instances with each type of relationship follow.

Construction of Instances with a Many-to-Many Relationship

A relation in the ACOVER has a many-to-many relationship between its attributes if there is no FD in the ZCOVER associated with that relation. An instance of such a relation, say R(A,B,...),

R(A)	S(B)	T(A,B)
A1	B1	A1 B1
A2	B2	A1 B2
A3	B3	A1 B3
		A2 B1
		A3 B1

Figure 2. Instance with Many-to-Many Relationship between A and B

can be generated by creating an instance of R(A,B) with a many-to-many relationship between A and B. And then, creating an instance of R(A,B,C) with a many-to-many relationship between A,B and C. And then, an instance of R(A,B,C,D) with a many-to-many relationship between A,B,C and D, etc. A many-to-many relationship can be created in the following manner as described in [26].

"Given a relation R(X) with a many-to-many relationship among the attributes in X and a unary relation S(A), a relation T(X,A) with a many-to-many relationship among all attributes in X and A is constructed by concatenating the first tuple of R(X) with all tuples of S(A) and the remaining tuples of R(X) with the first tuple of S(A)"[26, page 121].

For example, an instance of R(A,B,C) with a many-to-many relationship among its attributes can be constructed as follows. Let the domain of A (denoted DOM(A)) = {A1, A2, A3}, DOM(B) = {B1, B2, B3}, and DOM(C) = {C1, C2, C3}. Then, the instance will be generated in two steps. First, an instance of T(A,B) will be created with a many-to-many relationship among its attributes as shown in Figure 2.

Then, an instance of T(A,B,C) will be created as shown in Figure 3.

Construction of Instances with a One-to-One Relationship

An instance of a relation with a one-to-one relationship is generated from two relations consisting of the attributes contained in the LHS and RHS of the FDs which hold on the relation. That is, if the relation R(X,Y) has FDs X→Y and Y→X, then the instance is generated from the two

R(A,B)	S(C)	T(A,B,C)
A1 B1	C1	A1 B1 C1
A1 B2	C2	A1 B1 C2
A1 B3	C3	A1 B1 C3
A2 B1		A1 B2 C1
A3 B1		A1 B3 C1
		A2 B1 C1
		A3 B1 C1

Figure 3. Instance with Many-to-Many Relationship between AB and C

relations R(X) and S(Y), where X and Y are sets of attributes. If X contains only one attribute, then R(X) is a unary relation which contains all values from the domain of the attribute in X. If X contains more than one attribute, then R(X) is generated by creating an instance of a relation with a many-to-many relationship between the attributes in X. The relation S(Y) is also created in this same way. The instance of the relation with a one-to-one relationship among its attributes, T(X,Y), is created from the two relations R(X) and S(Y) in the following manner as defined in [26].

1. Set T(X,Y) to the empty set. Remove the first tuple of R(X) and the first tuple of S(Y), concatenate them and insert the resulting tuple into T(X,Y).
2. If $|X| = 1$ or $|Y| = 1$ repeat step 3 until R(X) and S(Y) become empty. If $|X| > 1$ and $|Y| > 1$, for each attribute A in X and each attribute B in Y execute step 4. If some tuples still remain in R(X) and S(Y), repeat step 3 until R(X) and S(Y) become empty.
3. Remove a tuple of R(X) and a tuple of S(Y), concatenate them and insert the resulting tuple into T(X,Y).
4. If T(X,Y) does not contain two tuples with A-values = A1 (first value in DOM(A)) but different B-values, execute step 5. If T(X,Y) does not contain two tuples with different A-values but with B-values = B1 (first value in DOM(B)), execute step 6.
5. Remove a tuple from R(X) with A-value equal to A1 and a tuple from S(Y) with B-value different from B1, concatenate them, and insert the resulting tuple into T(X,Y).
6. Remove a tuple from R(X) with A-value different from A1 and a tuple from S(Y) with B-value equal to B1, concatenate them, and insert the resulting tuple into T(X,Y).

The following example from [26] shows the generation of a relation with a one-to-one relationship among its attributes. Let X = {A,B,C} and Y = {D,E}, DOM(A) = {A1,A2,A3,A4},

R(A,B,C)	S(D,E)	T(A,B,C,D,E)
A1 B1 C1	D1 E1	A1 B1 C1 D1 E1
A1 B1 C2	D1 E2	A1 B1 C2 D2 E1
A1 B1 C3	D1 E3	A2 B1 C1 D1 E2
A1 B2 C1	D1 E4	A1 B1 C3 D1 E3
A1 B3 C1	D2 E1	A3 B1 C1 D3 E1
A2 B1 C1	D3 E1	A1 B2 C1 D1 E4
A3 B1 C1	D4 E1	A1 B3 C1 D4 E1
A4 B1 C1	D5 E1	A4 B1 C1 D5 E1

Figure 4. Instance with One-to-One Relationship between ABC and DE [26]

DOM(B) = {B1,B2,B3}, DOM(C) = {C1,C2,C3}, DOM(D) = {D1,D2,D3,D4,D5}, and DOM(E) = {E1,E2,E3,E4}. Figure 4 shows the construction of an instance of T(A,B,C,D,E) with a one-to-one relationship between {A,B,C} and {D,E}.

Construction of Instances with Many-to-One Relationships

Instances with many-to-many relationships are generated from two relations R(X) and S(Y) which are constructed the same way as described above for one-to-one relationships. The relation T(X,Y) is created as follows [26, page 124]:

1. Concatenate the first tuple of R(X) to the first tuple of S(Y).
2. For each attribute A in X (from right to left) do the following, where s is $|S(Y)|$, a is $|DOM(A)|$ (we must have a>s), and A1 is the first value in DOM(A).
   (a) Concatenate the first s−1 tuples of R(X) with A-value different from A1 to the second to sth tuples of S(Y).
   (b) Concatenate the a − s remaining tuples of R(X) with A-value different from A1 to the first tuple of S(Y).

For example, if the relation R(A,B,C,D) has a many-to-many relationship between {A,B} and {C,D}, the instance can be constructed as follows. Let DOM(A) = {A1,A2,A3,A4}, DOM(B) = {B1,B2,B3,B4}, DOM(C) = {C1,C2}, and DOM(D) = {D1,D2}. An instance showing the many-to-one relationship is shown in Figure 5.

If a relation R(X,Y) has more than one FD associated with it, then one of the FDs must have a scope of (X,Y), and the additional FDs Z → W are of the form $Z \subset X$ and $W \subset Y$ [26]. This

R(A,B)	S(C,D)	T(A,B,C,D)
A1 B1	C1 D1	A1 B1 C1 D1
A1 B2	C1 D2	A1 B2 C1 D2
A1 B3	C2 D1	A1 B3 C2 D1
A1 B4		A1 B4 C1 D1
A2 B1		A2 B1 C1 D2
A3 B1		A3 B1 C2 D1
A4 B1		A4 B1 C1 D1

Figure 5. Instance of Many-to-One Relationship between AB and CD

T(A,B,C,D)
A1 B1 C1 D1
A1 B2 C1 D2
A1 B3 C1 D1
A1 B4 C1 D1
A2 B1 C1 D2
A3 B1 C2 D1
A4 B1 C1 D1

Figure 6. Instance of Many-to-One Relationship between AB and CD, and A and C

is due to the rearrangement procedures described in Section 4.2.3.2. To generate the instance in this case, an instance is first constructed which shows the FD which has scope (X,Y). Then, the instance is modified for the other FDs in the following manner. "In every tuple where, for each attribute A ∈ Z, the A-value is equal to A1 (that is, the first value in DOM(A)), for each attribute B ∈ W the B-value is changed to B1 (that is, the first value in DOM(B))"[26, page 125]. For example, in the instance shown in Figure 5, the FD with scope (X,Y), i.e. AB → CD, is already represented. Then, if the FD A → C also holds on this relation, the instance will be modified as shown in Figure 6.

In the modified instance, only the third tuple needed to be changed. This is because the A-value was A1, so the C-value which was C2, had to be changed to C1. All other tuples with A-value = A1 already had a C-value = C1.

Construction of Instances with Combined Relationships

"An instance T(X,Y,Z) showing a combined relationship is constructed from the relations R(X,Y) and S(Z). We construct R(X,Y) by adding to an instance with a many-to-many relationship between X and Y a tuple obtained by concatenating the last tuples of P(X) and Q(Y) which are the relations corresponding to X and Y. If $|X| = 1$, then P(X) is the unary relation constructed from the domain of the attribute in X. If $|X| > 1$, then P(X) is obtained by generating a relation with a many-to-many relationship among the attributes in X. In the same way we obtain Q(Y) and S(Z). Let p be $|P(X)|$, q be $|Q(Y)|$, and s be $|S(Z)|$. We must have $p \geq 3$, $q \geq 3$ and $s = p + q - 3$."[26, page 126]

T(X,Y,Z) is created as follows [26, page 126]:

1. Concatenate the first tuple of R(X,Y) with the first tuple of S(Z).
2. Concatenate tuples 2, 3, ...,q−1 of R(X,Y) with tuples 2, 4, ..., s−1 of S(Z) respectively.
3. If there exists an MVD $Z \twoheadrightarrow Y$ in the ZCOVER. concatenate tuple q of R(X,Y) with tuple 1 of S(Z).
4. Concatenate tuples q+1, q+2, ..., q+p-2 of R(X,Y) with tuples 3, 5, ..., s of S(Z) respectively.
5. If there exists an MVD $Z \twoheadrightarrow X$ in the ZCOVER, concatenate tuple p+q-1 of R(X,Y) with tuple 1 of S(Z).
6. If there exist MVDs $Z \twoheadrightarrow X$ and $Z \twoheadrightarrow Y$ in the ZCOVER, concatenate the last tuple of R(X,Y) with tuple 1 of S(Z).

For example, let the FD $AB \rightarrow C$ and the MVDs $C \twoheadrightarrow A$ and $C \twoheadrightarrow B$ hold on the relation T(A,B,C). Additionally, let DOM(A) = {A1,A2,A3}, DOM(B) = {B1,B2,B3}, and DOM(C) = {C1,C2,C3}. Then, a combined relationship exists among attributes A,B, and C. Construction of an instance of T(A,B,C) showing this relationship is shown in Figure 7.

4.2.3.6 Compute Join of Relations and Display to User. As shown in the driver for the Armstrong relation function (see Section 4.2.3), after an instance of each relation is generated, the instances are joined on equal attributes (that is, the natural join of the relations is computed) to create the instance of the Armstrong relation. An example of this join operation is shown in

S(A,B)	R(C)	T(A,B,C)
A1 B1	C1	A1 B1 C1
A1 B2	C2	A1 B2 C2
A1 B3	C3	A1 B3 C1
A2 B1		A2 B1 C3
A3 B1		A3 B1 C1
A3 B3		A3 B3 C1

Figure 7. Instance with Combined Relationship between ABC

Section 4.2.3.7. After the instance is complete, the final step in this process is to display the results to the user.

4.2.3.7 Example of Entire Construction Process. The following example, which is based on an example presented in [26], shows the entire process required to generate an instance of an Armstrong relation.

INPUT:

UNIVERSAL RELATION:

R(A,B,C,D,E,F)

FDs and MVDs which hold on the UNIVERSAL RELATION:

$A \rightarrow B$

$A \twoheadrightarrow CD$

$A \twoheadrightarrow EF$

$AC \rightarrow D$

$E \rightarrow F$

$E \twoheadrightarrow ABCD$

Step 1. Decomposition:

Decomposition of the universal relation is done using the algorithm described in Section 4.2.3.1. The algorithm yields the following output.

ACOVER	*ZCOVER*
1. R(A,B)	1. $A \rightarrow B$
2. R(A,C,D)	2. $AC \rightarrow D$
3. R(E,F)	3. $E \rightarrow F$
4. R(A,E)	

Step 2. Rearrangement:

Relations 1, 2, and 3 of the *ACOVER* contain many-to-one relationships as shown by the corresponding FDs in the *ZCOVER*. Relation 4 contains a many-to-many relationship since there is no FD associated with it in the *ZCOVER*. There is no rearrangement necessary for relations with many-to-many relationships, therefore relation 4 does not need to be modified. However, the rearrangement procedures described in Section 4.2.3.2 for many-to-one relationships must be applied. Since none of the FDs in the *ZCOVER* have the same LHS, the first step of the rearrangement procedure does not apply. That is, there are no equivalent keys to merge. The second step of the procedure will discover that the *ZCOVER* contains the FDs $AC \rightarrow D$ and $A \rightarrow B$. Then, since $A \subset AC$, the relation $R[AC, (D \cup B)]$, i.e., R(A,B,C,D), is added to the *ACOVER*, the associated FDs $AC \rightarrow D$ and $A \rightarrow B$ are added to the *ZCOVER*, and the relations R(A,B) and R(A,C,D) are deleted from the *ACOVER*. This rearrangement step ensures that all attributes determined by a particular set of attributes, in this case attribute A, are in a single relation. The rearrangement procedures yield the following output.

Attribute	Domain Size	Domain Elements
A	4	A1,A2,A3,A4
B	2	B1,B2
C	4	C1,C2,C3,C4
D	2	D1,D2
E	3	E1,E2,E3
F	2	F1,F2

Figure 8. Domain Sizes and Domain Elements

ACOVER	*ZCOVER*
1. R(A,B,C,D)	1. AC → BD
	1. A → B
3. R(E,F)	3. E → F
4. R(A,E)	

Step 3 and Step 4. Compute Domain Sizes and Store Attributes in Unary Relations:

The minimum domain size for each attribute is computed using the algorithm shown in Section 4.2.3.3, and then the domain elements are created and stored in a unary relation corresponding to each attribute. The output of these procedures is shown in Figure 8.

Step 5. Construct Instances:

Step 5.1. Instance for Relation 1:

Relation 1 contains two many-to-one relationships as represented by the corresponding FDs in the *ZCOVER*. The instance for this relation, shown in Figure 9 is generated by the algorithm described in Section 4.2.3.5 for many-to-one relationships.

T(A,C,B,D)
A1 C1 B1 D2
A1 C2 B1 D1
A1 C3 B1 D2
A1 C4 B1 D2
A2 C1 B1 D1
A3 C1 B2 D2
A4 C1 B1 D2

Figure 9. Instance of Relation 1

T3(E,F)
E1 F1
E2 F2
E3 F1

Figure 10. Instance of Relation 3

Step 5.2. Instance of Relation 3:

This relation also contains a many-to-one relationship as represented by the corresponding FD in the *ZCOVER*. The instance for this relation, shown in Figure 10, is also generated by the algorithm described in Section 4.2.3.5 for many-to-one relationships.

Step 5.3. Instance of Relation 4:

This relation has a many-to-many relationship since there is no associated FD in the *ZCOVER*. The instance for this relation, shown in Figure 11, is generated by the algorithm described in Section 4.2.3.5 for many-to-many relationships.

Step 6. Compute Join:

Computing the join of T1, T3, and T4 on equal attributes produces the relation shown in Figure 12. This is an instance of an Armstrong relation for the given input set.

T4(A,E)
A1 E1
A1 E2
A1 E3
A2 E1
A3 E1
A4 E1

Figure 11. Instance of Relation 4

T431(A,B,C,D,E,F)
A1 B1 C1 D2 E1 F1
A1 B1 C1 D2 E2 F2
A1 B1 C1 D2 E3 F1
A1 B1 C2 D1 E1 F1
A1 B1 C2 D1 E2 F2
A1 B1 C2 D1 E3 F1
A1 B1 C3 D2 E1 F1
A1 B1 C3 D2 E2 F2
A1 B1 C3 D2 E3 F1
A1 B1 C4 D2 E1 F1
A1 B1 C4 D2 E2 F2
A1 B1 C4 D2 E3 F1
A2 B1 C1 D1 E1 F1
A3 B2 C1 D2 E1 F1
A4 B1 C1 D2 E1 F1

Figure 12. Instance of an Armstrong Relation for Example Input

Since all of the algorithms described throughout Section 4.2.3 are required to create an instance of an Armstrong relation, they were selected for implementation in the toolbox to support the Armstrong relation function. The last algorithm needed for the toolbox, an alternative design algorithm, is discussed in the next section.

4.2.4 Alternative Logical Designs. A literature review was conducted to determine if any algorithms had been published to automatically generate *all* possible alternative 3NF database schemes for a given set of attributes and functional dependencies. Although the literature review did not reveal any published algorithms, many authors have pointed out that alternative 3NF schemes can be generated with the various decomposition algorithms by:

1. Varying the order of the attributes on the LHS of FDs prior to the left reduction procedure described above [3]. This can result in different designs since the RHS may be functionally dependent on two different sets of attributes. For example, if the set of FDs given to hold on a database were:

   $$A\ B \rightarrow D$$

   $$A \rightarrow C$$

   $$C \rightarrow D$$

   $$B \rightarrow E$$

   $$E \rightarrow D$$

   then, both A → D and B → D are implied. Thus, if A is removed from the LHS of A B → D first, then since B functionally determines D, A will be considered extraneous and therefore permanently removed from the LHS leaving B → D. However, if B is removed from the LHS first, since A → D, B will be permanently removed from the LHS leaving A → D. Since each FD will ultimately be incorporated into a separate scheme, the order in which attributes are removed from the LHS of FDs can potentially effect the final database design.

2. Varying the order of the FDs which hold on the database attributes prior to removing redundant FDs can also effect the final design. Different orders of FDs have the potential to generate different database designs for a similar reason to that of varying the order of the LHS attributes described above. That is, removing certain FDs first can effect whether or not other FDs are subsequently removed [3].

Thus, one approach to generating alternative database schemes could be based on varying the order of the LHS attributes and varying the order of the given FDs, such as in the following algorithm.

INPUT: Set of universal attributes and a set F of FDs which hold over those attributes.
OUTPUT: Set of alternative 3NF schemes.
n_i = number of attributes on the LHS of each FD where i equals
    1 to m and m is the number of FDs in the input set.

```
begin
    Step 1: Compute the minimal cover of the input set of FDs;
    Step 2: Compute the closure of each LHS;
    Step 3: Right reduce the FDs;
    Step 4: /* comment: generate designs */
    For $i = 1$ to $m$
        begin
            For each different order of the LHS attributes of $FD_i$
                begin
                    For each different order of the FDs in F
                        begin
                            Compute 3NF schemes;
                        end
                end
        end
end
```

The first step of the algorithm is required to reduce redundancy in the FDs which cannot effect alternative designs. For example, the step will remove trivial dependencies and extraneous attributes from the LHS of FDs which, if not removed at the beginning of the algorithm, must be removed each time the 3NF algorithm is executed.

Then, Steps 2 and 3 are required because the 3NF design algorithm uses a minimal cover algorithm which always produces a minimal cover that is a subset of the input set of FDs [3]. Thus, the 3NF designs will always be based on subsets of the input set of FDs unless the the algorithm first expands the set of FDs to the set of all FDs logically implied by F which have the potential to effect alternative designs. The algorithm could compute the closure of the set of FDs, that is, the set of all FDs logically implied by F by applying the following rules which are known as Armstrong's axioms [16]:

- Reflexivity rule. If X is a set of attributes and $Y \subseteq X$, then $X \rightarrow Y$ holds.
- Augmentation rule. If $X \rightarrow Y$ holds and W is a set of attributes, then $WX \rightarrow WY$ holds.
- Transitivity rule. If $X \rightarrow Y$ holds, and $Y \rightarrow Z$ holds, then $X \rightarrow Z$ holds.

However, computing the closure is not necessary because it generates many FDs which will not effect the final schemes. For example, all of the FDs generated by the reflexivity rule are trivial FDs since the RHS is a subset of the LHS, and thus these FDs will always be removed from F by the minimal cover algorithm when it deletes redundant FDs. Thus, these FDs will never effect the final scheme. Therefore, Step 2, using the attribute closure algorithm defined in Section 4.2.2, will compute all the FDs implied by the LHS attributes of each FD using the augmentation and transitivity rules, and will thus be sufficient to generate all FDs logically implied by F which have the potential to generate different alternative database schemes.

Step 4 generates the schemes; however, this part of the algorithm has the potential to be prohibitively time consuming. That is, the number of designs generated will become quite large as the number of FDs increases and the number of attributes on the LHS of each FD increases. Specifically, the execution time of line 6 is directly related to $m!$ since there are $m!$ possible orders of m FDs, and the execution time of line 5 is directly related to $n_i!$ since there are $n_i!$ possible orders of n_i attributes in the LHS. Then, since line 6 is inside a loop controlled by line 5, which will be executed $n_i!$ times, the two loops will execute a number of times on the order of $(\sum_{i=1}^{m} n_i!)m!$. The execution time of an algorithm which performs a factorial × factorial number of operations will increase very rapidly as the input sets grow larger. For example, if the input set consists of 3 FDs where each LHS of the FDs contains 3 attributes, the loops would execute $3! \times 3! = 36$. But, if the input set consists of 6 FDs where each LHS contains 6 attributes, the loops would execute 6! × 6! = 518,400 times. Then, likewise, 7! × 7! = 25,401,600 times. Obviously, as n and m increase, the algorithm will generate an extremely large number of database designs and could potentially take a significant amount of time to finish. Another problem which must be considered is that many of the designs generated would be identical since varying the orders of the LHS attributes and the FDs will only produce different designs in some cases. Therefore, much of tne excessive time and space requirements of the algorithm would be wasteful.

Due to the problems with this approach, it was obvious that the number of database designs generated must be reduced. Since many of the designs generated would be identical, one way to reduce the number of database designs would be to never generate the identical schemes. This approach requires that the algorithm include some type of "rules", i.e. heuristics, so that only the orders of FDs and LHS attributes which will cause different designs are tried. Thus, in order to determine what the "rules" for the algorithm must be, the toolbox had to include a function which could support the research in this area.

Since the generation of alternative designs is based on varying the order of LHS attributes and FDs, the toolbox function needed to allow the user to vary these factors so the resulting effects on the designs could be analyzed. However, since the number of attributes on the LHS of an FD will typically be small (e.g., 1-3), the majority of different designs will most likely be a result of varying the order of the FDs rather than the LHS attributes. Thus, the toolbox was designed so that the 3NF design function includes an alternative design option which allows the user to vary the order of the FDs to study the effect on the final design. The function is designed so it can later be enhanced so the user can vary the LHS attributes. Once the heuristics are determined, they could either be incorporated into an algorithm which automatically generates all alternative designs, or incorporated into an interactive user interface which provides the user with guidance on ways to generate other designs.

The BCNF/4NF design algorithm used in the toolbox can also be used to generate alternative designs; however, the algorithm generates database schemes based on MVDs rather than FDs as in the 3NF algorithm. Thus, the heuristics for an algorithm to generate alternative BCNF and 4NF schemes will most likely be different than those for alternative 3NF designs, and therefore, research should be conducted to determine the heuristics for both of these cases.

4.3 Data Structures and Files

4.3.1 Data Structures. The first step in selecting the data structures for the toolbox was to analyze the algorithms to determine the type of operations done most often. Then, based on those operations, determine the type of data structures which could support those operations most efficiently. For example, if the algorithms frequently perform searches, an efficient data structure would be a binary search tree since "a binary search tree provides the flexibility of a linked list and allows quicker access to any node than a linked list"[23]. However, if the algorithms typically look at every item in a list, then every node must be visited and therefore a binary search tree would provide no advantage over a linked list structure, and thus the overhead of building and maintaining the search tree would be wasteful.

Analysis of the algorithms selected for implementation in the toolbox revealed that, in general, the algorithms frequently must step through every dependency to accomplish their specific function. For example, the algorithm which generates the minimal cover of a set of MVDs must step through every dependency to eliminate trivial dependencies, right reduce, left reduce, and then eliminate redundant dependencies. The algorithm to compute the envelope set of a set of FDs and MVDs also must step through each dependency. Additionally, many of the algorithms which support the Armstrong relation function also examine every dependency. Another type of operation that occurs frequently in the algorithms is insertion and deletion of dependencies in lists of dependencies.

The analysis of algorithms also revealed that sorting the dependencies and searching for a specific dependency do not occur frequently. However, the membership algorithm performs operations which require quick access to specific dependencies, and the Armstrong relation function requires unique data structures for storage of relations.

Thus, it was determined that the universal attributes, the FDs, and the MVDs should be stored in linked list structures so algorithms could easily step through each one. Although array structures were considered, they were not selected for storing the FDs, MVDs, and attributes because the size of the input sets will be dynamic and linked lists are better suited for operations which require dynamic memory allocation. Additionally, linked lists are better structures for frequent insertion and deletion operations, and are therefore a better choice of data structures than arrays for the required operations. Since the membership algorithm and the Armstrong relation function require unique data structures, the structures for these functions can be built from attribute and dependency data stored in the linked lists.

For the membership algorithm, it is critical that the data structures support quick access to specific dependencies. The general data structures needed to support the algorithm are defined in [3]. Since this algorithm will be used extensively in the toolbox, it is critical that it executes fast. Therefore, the data structures used by the algorithm should mainly be arrays so the algorithm can use indexing to directly access specific items. Linked lists or trees would require searching, so the algorithm would take longer to access specific items.

For the Armstrong relation function, a linked list structure is best suited for storage of the relations. This is because most of the operations on the relation list involve stepping through each relation, inserting relations in the list, and deleting relations from the list. Each relation structure will contain a pointer to a list of tuples. The Armstrong relation function performs many operations on tuples such as concatenating tuples and joining tuples on equal fields which require that the program know exactly where to find the value of each attribute in the tuple. Therefore, each tuple is stored as an array so the function can index directly to each attribute value.

4.3.2 Files

4.3.2.1 Input Files. As stated in Chapter III, one of the long range objectives of the toolbox is to serve as a normalization tool in a suite of stand-alone database design tools developed at AFIT. Therefore, the file formats used by the toolbox must be designed to provide a standard interface so database attributes and constraints can be passed between all tools.

The method chosen to accomplish this objective was to define the format for text files which will be input into the toolbox. Then, if a tool such as an ER diagramming tool wants to "interface" with the toolbox, it would simply have to store its output in a text file in the proper format so the toolbox could read it. If the ER diagramming tool only stored a set of universal attributes in the file, then the user could add constraints (dependencies) to the file through file update functions in the toolbox.

The specific format of the input files used by the toolbox was originally defined in [17]. The file format was adopted for use in the toolbox for several reasons. For one thing, the format is easy to read and understand. Additionally, [17] had already defined the grammar and syntax of the input files for the UNIX *lex* and *yacc* functions. The *lex* function, or lexical scanner, is used to verify the grammar of the input file and find tokens for the syntax parser *yacc*. *Yacc* is used to verify the syntax of the input files, notify the user of errors in the input file, and store the universal attributes, FDs, and MVDs in linked list structures. Thus, the file format from [17] was adopted, and the parser (*yacc*) is used to identify format errors when files are created, updated, or files are named as input files for a toolbox function. Examples of the file format are shown in Appendix D.

4.3.2.2 Output Files. The output of the 3NF, 4NF, BCNF, and Armstrong relation functions are displayed on the screen and are also written to files for later reference. To help the user identify these files, the system names them by using the input filename plus one of the following extensions: 3NF, 4NF, BCNF, arm, respectively. Examples of the formats and sample output are shown in Appendix D.

4.4 User Interface

The user interface was designed for the target user defined in Chapter III. That is, the interface is designed for users who are familiar with dependency theory and normalization concepts. Therefore, the interface frequently uses technical terms which assume a working knowledge of dependency theory.

Since user interface issues were not central to the goals of this thesis, a simple menu system was adopted. The interface is designed so the user will need minimal assistance from the user's manual to operate the system. The menus present all available options at each level, and they allow the user to return to the previous menu. The menus provide the system with a structured interface so the user always knows what the options are, and how to get back to the main menu. User interface issues were not central to the goals of this thesis, therefore, a simple menu system was adopted.

Keyboard input was the only practical choice for data input capability since the users will be working from a wide variety of terminal types, with varying input capability.

V. Coding and Implementation

5.1 Hardware Configuration

The toolbox was developed and implemented on the ICC (Interim Computer Capability) at AFIT for two critical reasons. First, since the toolbox will be used primarily as a teaching aid and a research tool at AFIT, the program can be utilized by more users if installed on one of the school's centralized computer systems. Second, since the toolbox includes many algorithms which have a time complexity which is related to the size of the input set of universal attributes and dependencies, the program needed to be implemented on the centralized system which executes the most instructions per second. Therefore, the ICC was best suited for this application.

5.2 Language Selection

The toolbox was written in the C programming language for several reasons. First, C is well suited for this type of application because it compiles into efficient executable code. Efficiency is an important characteristic for the toolbox since the execution time of most of the algorithms is a function of the number of attributes and dependencies in the input file, and thus, the algorithms could potentially require relatively long execution times. Additionally, C code can easily be moved to and compiled on other AFIT systems. For example, the toolbox modules can be transferred to the ASC (Academic Support Computer) and the SSC (Scientific Support Computer) in the exact same form as they are on the ICC. Also, the code can be run on the LSI-11's with minor modifications. Finally, C was chosen because two readily available database design tools [17,19] were written in C, and therefore, some of the routines and algorithms from those tools could be directly incorporated into the toolbox if the toolbox was written in C. Using C as the programming language, therefore, had the potential to reduce development time by several months. Thus, based on the above reasons, C offered many benefits, and was therefore chosen as the programming language.

5.3 Coding

The software for the toolbox was written in a top-down, structured fashion. That is, the top level modules were written first, with lower level functions stubbed out. Then, as development progressed, each of the progressively lower level functions were completed, and then finally the lowest level functions were completed. The program modules were organized into files based on function. For example, the algorithms required to support the Armstrong relation function are in one file, the algorithms required to support the 4NF/BCNF functions are in a separate file, the code which drives the menus and the function interfaces is in another, etc. Organizing the modules into files based on function will enhance the maintainability of the toolbox software.

VI. Acceptance Testing

This chapter documents the testing phase of the toolbox development.

6.1 Scope of Testing

The main objectives of the testing phase focused on the following main software engineering concepts:

- **verification** - verify that the product is built right based upon requirements.
- **validation** - validate that the code does what the user wants. That is, ensure we built the right product.

Thus, the testing approach was based on the following specific objectives:

1. Ensure all toolbox functions produce accurate results.
2. Ensure the toolbox satisfies user needs.

6.2 Test Plan

Based on the specific test objectives, testing was divided into the following two phases:

Phase I Function Accuracy.

Phase II Operational Needs.

For the first testing phase, the test objectives could be met by creating specific test cases which had a given input and a known output, and then comparing the output from each function to the expected output. The input for the test cases had to be designed to represent a wide variety of possible cases. Thus, a test procedure was set up for each toolbox function with the required input and expected output specified for each test case. Each test procedure is documented in Appendix D.

For the second testing phase, the testing criteria were not as clear cut as for the first phase. That is, for the second phase, there were no clear lines between correct results versus incorrect results. Thus, the approach taken to test whether operational needs were satisfied focused on whether the toolbox accomplished the specific tasks specified in the user requirements documented in Chapter III. And also, focused on an analysis of whether or not the toolbox had the potential to accomplish the more broad tasks required by the user (i.e., serving as a research tool and a teaching aid).

The results of the tests required for the first testing phase, and the analysis required for the second testing phase are presented in Section 6.4.

6.3 Test Procedures

Each toolbox function was tested by inputting given input data into the function, and then comparing the actual toolbox output to the expected output. The input and expected output for each test are described in Appendix D.

6.4 Test Results

6.4.1 Results of Phase I. The actual output of the toolbox matched the expected output for each of the test cases defined in Appendix D. Thus, the toolbox satisfies the test criteria for Phase I of the testing phase.

6.4.2 Results of Phase II. An analysis indicated that the toolbox provides all the functions specified in the user requirements. Therefore, the toolbox meets the operational needs of the user from the perspective that it provides the functions necessary to accomplish the tasks which were defined in the requirements. Additionally, analysis showed that the toolbox has the potential to accomplish the broader tasks defined in the requirements (i.e., serving as a research tool and a teaching aid).

The toolbox can be used as a research tool in several ways. For example, the user can vary the database specifications which are input into the various toolbox functions and study the impact on the function output. Also, the toolbox provides a very good capability for the user to study and research alternative logical designs since the 3NF design function allows the user to vary the order of the FDs, and thereby create alternative 3NF designs. The BCNF and 4NF design functions also allow the user to study alternative designs by letting the user change the order of the keys for the schemes, thereby creating alternative designs. Thus, the user could analyze which order changes actually create different designs. The effectiveness of the toolbox as a research tool depends on the objectives of the specific experiments done, and the design of the experiments, therefore, it is difficult to generalize about effectiveness. The toolbox would have to be examined in each specific situation to obtain specific results, however, the toolbox has potential to facilitate research in some areas.

The toolbox could also be used as a teaching aid in several ways. A student could study the output of the toolbox to learn the functions of the various database design algorithms, and to analyze the results of varying the input. Additionally, the Armstrong relation function in the toolbox could be used to teach students the concepts of functional dependency and multivalued dependency. The effectiveness of the toolbox as a teaching aid would again depend on the specific lesson plan, and the objectives of the lesson, and thus the toolbox would have to be examined in specific situations to obtain specific results.

Thus, the analysis indicates that the toolbox performs the functions required to fulfill the operational needs of the user; however, the actual effectiveness of the toolbox to serve as a research tool and a teaching aid depends on many subjective factors which will depend on the specific situation and purpose for which the toolbox is used. Therefore, based on the analysis, the toolbox satisfies the test criteria for Phase II of the testing phase.

VII. Conclusions and Recommendations for Further Study

7.1 Conclusions

Designing a database is a very time consuming and complex set of activities. Although researchers have investigated implementing computer tools to assist designers in all phases of database design, a significant effort has been applied in the area of logical design. This is because the process of logical design is well suited for computer assistance because the process can be time consuming, repetitive, and it can be structured into a clearly defined set of steps. Additionally, computer assistance is needed in this area because much of the current dependency theory used to design and study the logical structures of relational databases exists in the form of published algorithms and theorems, and hand simulating these algorithms can be a tedious and error prone chore. The literature review included in this study revealed that even though many researchers have developed computer tools to assist database designers with the logical design of relaticnal databases, there are still many algorithms and functions which need to be incorporated into automated design tools. Thus, the objective of this thesis investigation was to design and implement a computer tool which automates some of these algorithms and functions.

The computer tool, or "Dependency Theory Toolbox", was designed for use in an academic environment as a teaching aid and research tool, rather than for practical application to database design problems. The toolbox provides many functions which allow the user to generate and study database designs, and is specifically designed to support research in the area of alternative database designs. Much research is still needed in this area to define methods for automatically generating all alternative designs for a given set of universal attributes and FDs which hold over those attributes.

Many authors have pointed out that alternative 3NF schemes can be generated by varying the order of the attributes on the LHS of FDs prior to left reducing the FDs, and by varying the order of the FDs prior to removing redundant FDs. Thus, one approach to generating all alternative database schemes could be based on varying the order of the LHS attributes and varying the order of the given FDs. However, such an approach has the potential to generate a very large number of database designs since generating every possible order of n elements results in $n!$ orders. So, the number of orders, and thus, the number of designs, would increase very rapidly as the number of attributes and FDs increases. Another problem with this approach is that many of the schemes generated by this method would be identical since varying the orders of the LHS attributes and the FDs will only produce different designs in some cases. Thus, in order for this approach to be practical, a design algorithm based on this method must include some type of "rules", i.e. heuristics, so that only the orders of FDs and LHS attributes which will cause different designs are used as input. The toolbox was specifically designed to support research in this area by allowing the user to vary the order of FDs so the resulting effects can be studied. The 4NF and BCNF design functions also have similar functions.

The result of this thesis investigation was that the "Dependency Theory Toolbox" was successfully implemented, documented, and tested. The toolbox provides all the functions specified in the requirements analysis, with the limitation that it does not generate all alternative logical designs automatically. The toolbox does, however, include a function which allows the user to generate alternative designs manually by varying the order of FDs input into the 3NF function, and by varying the order of keys input into the BCNF and 4NF functions.

In conclusion, we see that even though a significant amount of effort has been invested in studying the logical structures of relational databases, and in developing automated database design aids, this area of database design still warrants further analysis and study.

7.2 Recommendations for Further Study

Although many database design algorithms were implemented in the "Dependency Theor, Toolbox", there are still many other algorithms and functions which need to be incorporated into automated design tools. For example, the toolbox could be expanded to include algorithms which determine if database schemes are dependency preserving, and if schemes can be joined without losing information (i.e, check for a lossless join).

Additionally, research should be done to define the heuristics needed to determine which orders of LHS attributes and FDs will cause different designs to be generated by the 3NF algorithm. The research in this area could ultimately lead to better database designs since a database designer could readily see all the available options, and thus choose the design best suited for the particular application.

Appendix A. *User's/Maintenance Manual*

A.1 Introduction

This User's/Maintenance Manual documents the capabilities incorporated in the Dependency Theory Toolbox, and explains how to use the functions provided. In general, the function of the toolbox is to assist the user with designing and studying the logical structures of relational databases and various related concepts of dependency theory. The system is intended for use in an academic environment as a teaching aid and research tool rather than for practical application to database design problems. However, the tool could be used to design small relational databases which have a limited number of attributes.

In order to use the functions in the toolbox, the user must first create an input file which contains database specifications. The toolbox allows the user to input database constraints (functional dependencies and multivalued dependencies) and database attributes interactively or to specify existing user files which contain the information. The user files can be created using the toolbox, or can be generated by other design tools in the prescribed format (defined below). The system is designed to accept existing files so that this toolbox can be used in conjunction with other database design tools developed at AFIT. For example, if the Entity-Relationship diagramming tool which was developed at AFIT, stored the entities and attributes which were specified in the diagram in text files with the same format, the data files could be directly input into the toolbox.

A.2 Toolbox Location

The toolbox was developed and implemented on the ICC (Interim Computer Capability) at AFIT. The ICC is an Elxsi 6400 computer. The toolbox is written in the C programming language, and was implemented under the UNIX operating system, version 4.2BSD. Currently, the executable code is stored on the SSC under the path name:

/course/course/ee646/dbtoolbox/dbtoolbox

A.3 Compiling and Linking

If the source code must be moved to a new directory, the information which defines all required files and system modules is documented at the beginning of file /course/course/ee646/dbtoolbox/main.c. The file header at the beginning of this file describes all files required to build the toolbox, and how to compile and link them.

The toolbox can be moved to other centralized AFIT systems which operate under the UNIX operating system. The source files can be moved by using the UNIX 'rcp' command, and then the executable program can be generated by compiling and linking the source code with the 'cc' command.

The file named "makefile", which is included with the source code files on the SSC in directory /course/course/ee646/dbtoolbox, contains all the specifications necessary to compile and link all toolbox modules using the "make" UNIX command. The contents of "makefile" are shown in Figure 13. The makefile defines all modules required to compile and link the toolbox, and shows which modules depend on the contents of other modules. For example, the file shows that main.o depends on the contents of header.h and global, and therefore, if these modules are changed, main.c must be recompiled to update main.o. This update, and any others, will automatically take place when header.h and global are changed by simply typing "make" at the UNIX prompt. Additionally, the makefile shows the UNIX commands required to compile each module.

A.4 Start-up Procedure

To start the toolbox, type the following path name from a directory on the SSC where you want the output files to be stored.

```
/course/course/ee646/dbtoolbox/dbtoolbox
```

```
toolbox: main.o lex.yy.o y.tab.o utility1.o utility2.o utility3.o fd_alg.o
                        BCNF_4NF_alg.o arm_rel.o
cc -g main.o lex.yy.o y.tab.o utility1.o utility2.o utility3.o
                        fd_alg.o BCNF_4NF_alg.o arm_rel.o -ll -o toolbox
main.o:  main.c header.h global
        cc -c -g main.c
y.tab.h:  yac
        yacc -d yac
y.tab.c:  yac
        yacc yac
lex.yy.c:  le y.tab.h
        lex le
lex.yy.o:  lex.yy.c y.tab.h header.h
        cc -c -g lex.yy.c
y.tab.o:  y.tab.c header.h
        cc -c -g y.tab.c
utility1.o:  utility1.c header.h
        cc  -c -g utility1.c
utility2.o:  utility2.c header.h
        cc  -c -g utility2.c
utility3.o:  utility3.c header.h
        cc -c -g utility3.c
fd_alg.o:  fd_alg.c header.h
        cc -c -g fd_alg.c
BCNF_4NF_alg.o: BCNF_4NF_alg.c header.h global
        cc -c -g BCNF_4NF_alg.c
arm_rel.o: arm_rel.c header.h
        cc -c -g arm_rel.c
```

Figure 13. Toolbox Makefile

If the path name */course/course/ee646/dbtoolbox* is in your path in your .login file, then the toolbox can be executed by simply typing *dbtoolbox* followed by a carriage return (denoted by -CR- in the rest of this manual). After you have typed "dbtoolbox -CR-", the system will display a welcome banner and tell you to press return when you want to continue.

After you type return, the system will display the main menu which contains the following options:

Main Menu

1. Create or Update Database Specification File
2. Generate Logical Structures
3. Accomplish Utility Functions
4. Exit Toolbox

This menu, like all other system menus, will be followed by the message:

Please type the number of your choice

In response to this message, you should type a number listed on the menu and then -CR- If you type a number which does not appear on the menu, the system will print an error message. and allow you to try again. Since all of the functions in the toolbox require an input file which contains database specifications, the user should choose option 1 the first time the toolbox is used This will allow him/her to create the required file. This is necessary because when the user chooses option 2 or 3, the system will ask for the input file name before executing the requested function The user may exit the system from the main menu or any of the menus below it The functions which are provided through the main menu are described in the following sections

A.5 Overview of User Interface

The user interface for the toolbox is menu driven. It is designed so the user will need minimal assistance from the user's manual to operate the system. The menus present all available options at each level, and they allow the user to return to the previous menu. The menus provide the system with a structured interface so the user always knows what the options are, and how to get back to the main menu.

Additionally, the interface is designed for users who are familiar with the process of relational database design, and with dependency theory concepts related to database design. Each of the utility functions provides a short description of the function's purpose prior to actually executing the function. However, the user will find the toolbox much more useful if they review concepts such as: minimal cover, attribute closure, envelope set, etc. either prior to using the toolbox or in conjunction with using the toolbox.

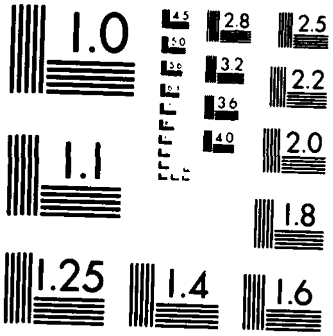
A.6 Main Functions

Once the user types the number of an option on the main menu, the system executes the appropriate function. Each function is described below.

A.6.1 Create or Update Database Specification File. This function allows the user to create or update an input file which contains a set of universal attributes and a set of functional dependencies (FDs) and/or multivalued dependencies (MVDs). The subsections below describe how the options within this function allow the user to create files in the prescribed format. However, the operation of all the functions follows the same basic format. First, the options request the names of the required file(s). Then, once the file is created or retrieved, the file is displayed and the user can input and edit data using a full screen editor using UNIX vi editor commands. After the user is done editing a file, the toolbox will examine the file to ensure it is in the proper format. If it is, the system will return to the "Create or Update Database Specification File me[illegible]

AD-A188 872 DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE 2/2
DESIGN(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB
OH C W STANSBERRY DEC 87 AFIT/GCS/ENG/87D-26

UNCLASSIFIED F/G 12/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

is not in the proper format, the toolbox will display the file contents and, in most cases, tell the user which line contains an error and what the error is. The system will then inform the user that he may edit the file and try again, and will return the user to the "Create or Update Database Specification" menu.

Some of the basic UNIX vi editor commands which can be used to create and edit input files are listed in Figure 14.

A.6.2 Input File Format. The input files must be created in the format shown in Figures 15 and 16. This format was originally defined in [17], and has been adapted for use in the toolbox. The following specifications define the format more precisely.

UNIVERSAL ATTRIBUTES:

The first data in the file must be a list of attributes followed by a period. The first attribute may be preceded by any number of blank lines, or no blank lines.

- Legal characters: A-Z, a-z, 0-9, and _ (i.e., underscore).
- Attributes must begin with a letter.
- Attributes can be strings of legal characters up to a maximum of 10 characters long.
- A maximum of 92 attributes can be specified in one file.
- Each attribute can be followed by any number of spaces or any number of carriage returns.
- List of attributes must end with a period. There can be any number of spaces, no spaces, or a carriage return between the last attribute and the ending period.

- **i** - insert text just before cursor.
- **I** - insert text at beginning of line.
- **esc** - escape out of insert mode.
- **a** - append text just after cursor.
- **A** - append text at end of line.
- **arrow keys** - move cursor direction of arrow.
- **control H** - backspace.
- **:n** - where n is an integer line number. This moves the cursor to line number n.
- **x** - delete the character under the cursor.
- **dd** - delete the entire line which the cursor is on.
- **o** - open a new line under the line where the cursor is.
- **O** - open a new line above the line where the cursor is.
- **:wq** - write the updated file over the old one, and quit the editor.
- **:q!** - quit, but don't save any of the changes.

Figure 14. Summary of UNIX vi Commands

```
A B C D E .
A – > B .
C – > E .
B – > – > C .
A – > E .
D – > – > A .
```

Figure 15. Input File Example 1

DATABASE DEPENDENCIES:

The list of attributes must be followed by at least one dependency. The file may contain any number of functional dependencies or multivalued dependencies. The dependencies must comply with the following specifications.

- The symbol to show functional dependence is – > (i.e., dash, greater than). The symbol must contain only one dash and one greater than.
- The symbol to show multivalued dependence is – > – >.
- Each dependency must have one or more legal attributes on the LHS and RHS, and must be followed by a period.
- The list of dependencies may contain embedded carriage returns and spaces.
- Multivalued dependencies may include the symbol "|" to separate groups of dependencies when appropriate on the RHS only.
- If all attributes in the attribute list contain only one symbol, then the attributes on the LHS and RHS do not need to be separated by spaces. However, if some of the attributes contain more than one legal character, each attribute must be separated by spaces or carriage returns.

```
NAME SSN ACCTNUM BALANCE .
SSN – > NAME .
SSN – > – > ACCTNUM BALANCE .
ACCTNUM – > BALANCE .
```

Figure 16. Input File Example 2

A.6.2.1 Create a new file. This function allows the user to create new files in the format defined in Section A.6.2. The function first asks the user for a file name for the new file. Then, the file is created and the user can enter data using UNIX vi commands (see Figure 14 for a summary of basic commands). The toolbox calls the UNIX vi editor to provide the edit capability, and therefore, all vi commands are available.

When the user has finished creating the new file, he should end the editing session with the vi command ":wq" to save the information. If the user types the vi command ":q!" at the end of the session, the data will not be saved. After the user is done editing a file, the toolbox will examine the file to ensure it is in the proper format. If it is, the system will return to the "Create or Update Database Specification File" menu. If the file is not in the proper format, the toolbox will display the file contents and, in most cases, tell the user which line contains an error and what the error is. The system will then inform the user that he may edit the file and try again, and will return the user to the "Create or Update Database Specification" menu and the user can either create/edit additional files, exit the toolbox, or return to the main menu so he can use other toolbox functions.

A.6.2.2 Update an existing file (save with same name). This function allows the user to update an existing file, and save the contents under the same file name. The function first asks for the name of the file to be updated. The system will check to make sure the file exists, and if so, will display the file and allow the user to edit it using the full screen editor. If the file does not exist, the system will notify the user and allow him to try again. The toolbox will examine the file to ensure it is in the proper format as described above for the "Create New File" function. Once the user finishes editing the file, the toolbox returns to the "Create or Update Database Specification

File" menu and the user can either create/edit additional files, exit the toolbox, or return to the main menu so he can use other toolbox functions.

Note: If during the editing session the user decides he wants to save the updated file under a new file name, he may end the editing session with the vi command "wq newfilename". This will store the updated information in a file named "newfilename", and leave the original file untouched.

A.6.2.3 Update an existing file (save with a different name). This function allows the user to update an existing file, and save the updated information under a new file name. The function first asks the user for the name of the file to be updated. Then, if the first file exists, the function asks for the new file name. The system then makes a copy of the original file and stores it in a file with the new name. Then the system displays the copy so the user can edit it. When the user terminates the editing session, the updated information will be stored in a file with the new file name. After editing is complete, the toolbox will examine the file to ensure it is in the proper format as described above for the "Create New File" function. Once the new file is completed by the user, the toolbox returns to the "Create or Update Database Specification File" menu and the user can either create/edit additional files, exit the toolbox, or return to the main menu so he can use other toolbox functions.

A.6.3 Generate Logical Structures. This function allows the user to generate logical database designs in 3NF, 4NF, or BCNF. As soon as the "Generate Logical Structures" option is chosen on the main menu, the toolbox will ask the user to input the name of an input file. The user must provide the name of an input file which contains database specifications in the format described in Section A.6.2. The system will then examine the file to ensure it is in the proper format. If the specified file is not in the proper format, the system will display the errors and then display the user's options. Once the name of a properly formatted input file is given, the toolbox shows the options on the "Generate Logical Structures" menu. The function of each of the options on this menu are described in the following subsections. The 4NF and BCNF designs are accomplished by a single algorithm which was originally developed in [17]. The algorithm has been adapted for use in the toolbox.

A.6.3.1 Generate 3NF Designs. The "Generate 3NF Designs" option generates 3NF database schemes from the universal attributes and FDs in the specified input file. Third Normal Form (3NF) is a normal form in which each relation of a database conforms to the following restrictions. First of all, each relation cannot contain nonkey attributes which are functionally dependent on part of the primary key for the relation. In other words, each nonkey attribute of each relation must be fully dependent on the primary key. (This requirement causes the relations to be in 2NF). Additionally, no relation can contain a nonkey attribute which is dependent on another nonkey attribute. This last requirement ensures that no nonkey attribute is transitively dependent on the primary key.

If the input file contains any MVDs, they are ignored by this function since they are not considered in 3NF designs. This function displays a menu with the following options:

1. Generate a single 3NF scheme.
2. Generate alternative schemes.
3. Return to Generate Logical Structures menu.
4. Change Input File Name.
5. Exit System.

If option number 1, Generate a single 3NF scheme, is chosen, the system will display the universal attributes, FDs, and MVDs in the current input file. If the file does not contain any FDs, the program will notify the user that the universal set of attributes is by default in 3NF. If the file contains FDs, the toolbox generates the 3NF schemes, and stores them in a file which has the extension ". 1_3NF" (i. e. , single 3NF) added to the name of the current input file name. After displaying the schemes on the terminal, the toolbox tells the user the name of the output file where the schemes are stored, and then returns to the "Generate 3NF Designs" menu.

If option number 2, Generate alternative schemes, is chosen, the system will display the universal attributes, FDs and MVDs as done in option 1. Again, if the file does not contain any FDs, the toolbox will notify the user that the universal set of attributes is by default in 3NF.

If the file contains FDs, the toolbox will display an enumerated list of the FDs, and let the user know that the displayed order is the default order which will be used to generate the 3NF design. The system then asks the user if he would like to change the order of the FDs, and if so, to type the number of the FDs in the desired new order. The user can generate different 3NF schemes by varying the order of the FDs since the order effects which FDs are eliminated from the set by the 3NF algorithm.

The system will allow the user to vary the order of the FDs as many times as desired, and all designs will be stored in a file which has the extension ". alt_3NF" (i. e. , alternative 3NF) added to the name of the current input file name. After displaying the schemes on the terminal, the toolbox tells the user the name of the output file where the schemes are stored, and then returns to the "Generate 3NF Designs" menu.

A.6.3.2 Generate BCNF Designs. The "Generate BCNF Designs" option generates BCNF database schemes from the universal attributes and FDs in the specified input file. Boyce/Codd Normal Form (BCNF) is a stronger normal form than 3NF. BCNF has the same restrictions as named above for 3NF, however, BCNF also requires that the LHS of each nontrivial functional dependency be a superkey of the relation it applies to. A superkey is a set of attributes which uniquely identifies each entity (tuple) of a relation. That is, a superkey functionally determines all attributes in the relation.

If the input file contains any MVDs, they are ignored by this function since they are not considered in BCNF designs. This function displays a menu with the following options:

1. Generate BCNF schemes.
2. Return to Generate Logical Structures menu.
3. Change Input File Name.
4. Exit System.

The following definitions from [17] are necessary for understanding the output of this function.

- *Envelope set*: set of MVDs which is logically implied by a set of FDs and MVDs.
- *Minimum cover of a set of MVDs*: a reduced set of MVDs which is equivalent to the original set, but with no redundancies.
- *Dependency Basis of a set of attributes*: the dependency basis of a set of attributes X is the set of sets of attributes logically implied by X with respect to a given set of FDs and MVDs.
- M^-: if D is a set of FDs and/or MVDs, and M is the minimum cover of the envelope set of D, then M^- is the set of MVDs given by $\{X \twoheadrightarrow W | X \twoheadrightarrow W$ is a reduced MVD in $M^+\}$.
- *keys*: LHS(M^-)
- *sp-ordering*: a sequence of elements $X_1, X_2, \ldots, X_n$ is a *sp-ordering* if:

  (1) $X_i \subseteq X_j$ implies $1 \leq i < j \leq n$, and
  (2) if D logically implies $X_j \rightarrow X_i$ but D does not
  logically imply $X_i \rightarrow X_j$,
  then $1 \leq i \leq j \leq n$.

If option number 1, Generate BCNF schemes, is chosen, the system will display the universal attributes, FDs, and MVDs in the current input file. If the file does not contain any FDs, the program will notify the user that the universal set of attributes is by default in BCNF. If the file contains FDs, the toolbox will display an enumerated list of the keys for this set of dependencies, and let the user know that the displayed order is the default order which will be used to generate the BCNF design. The system then asks the user if he would like to change the order of the keys, and if so, to type the number of the keys in the desired new order. The user can generate different BCNF schemes by varying the order of the keys since the order effects which FDs are eliminated from the set by the BCNF algorithm.

In addition to the keys and list of BCNF schemes, this function also displays some of the intermediate results of the BCNF algorithm. The system displays the envelope set, minimum cover, dependency basis, and M^-.

The system will allow the user to vary the order of the keys as many times as desired, and all designs will be stored in a file which has the extension ". BCNF" added to the name of the current input file name. After displaying the schemes on the terminal, the toolbox tells the user the name of the output file where the schemes are stored, and then returns to the "Generate BCNF Designs" menu.

The other options on the "Generate BCNF Designs" menu are self explanatory.

A.6.3.3 Generate 4NF Designs. Fourth Normal Form (4NF) is a normal form which is defined exactly like BCNF except that instead of functional dependencies, the 4NF definition uses multivalued dependencies. That is, the LHS of each nontrivial multivalued dependency must be a superkey of the relation it applies to.

The "Generate 4NF Designs" function operates the same as the "Generate BCNF Designs" function described above. The only differences in the functions is that the output is stored in a file which has the extension ". 4NF" added to the name of the current input file name. Also, this function will work without FDs in the input file since the 4NF design algorithm can use FDs or MVDs separately to design schemes, or it can integrate both FDs and MVDs to design the schemes. If the file contains only FDs, this function will generate the same schemes as the BCNF function.

A.6.4 Accomplish Utility Functions. This activity provides many functions which the user can use to study dependency theory and relational database design. The user can select this function from the main menu. Prior to executing any of the functions, the user must provide the name of an input file which contains database specifications in the format described in Section A.6.2. After the name of the input file is provided, the toolbox will examine the file to determine if it is in the proper format. If not, the system will notify the user of the errors. If the file is in the proper format, the toolbox will display the following menu:

1. Find minimal covers for a set of FDs.
2. Find minimal covers for set of FDs and MVDs.
3. Membership algorithm.
4. Find envelope set for set of FDs and MVDs.
5. Compute attribute closure.
6. Find dependency basis of set of attributes.
7. Generate instance of an armstrong relation.
8. Return to Main Menu.
9. Change Input File Name.
10. Exit System.

The algorithms used to implement functions 2, 4, and 6 were originally defined in [17], and were adapted for use in the toolbox. The operation of each function is described in the subsections below.

A.6.4.1 FD Minimal Cover. A minimal cover of a set of FDs is a reduced set of FDs which is equivalent to the original set, but with no redundancies. The concept of minimal cover is central to normalization. Minimal covers are important because the cover contains all the same "potential" information as the original set; however, since the redundancies have been removed, the relations generated from a minimal cover should contain less redundancy.

This function first shows the following menu:

1. Generate minimal cover.
2. Return to Utility Function menu.
3. Change Input file Name.
4. Exit System.

If option number 1, Generate minimal cover, is chosen, the system will display the FDs in the current input file. Then, the minimal cover is displayed on the terminal until the user presses return. When the user presses return, the toolbox returns to the "FD Minimal Cover menu". The other options are self explanatory.

A.6.4.2 Combined FD/MVD Minimal Cover. As described above for FDs, the minimal cover of a set of MVDs is a reduced set of MVDs which is equivalent to the original set, but with no redundancies. This function operates the same as the FD minimal cover function except that it combines the FDs and the MVDs given in the input file into an equivalent set of MVDs, and then generates the minimal cover of the set of MVDs.

A.6.4.3 Membership Algorithm. Given an input file which contains a set of universal attributes and a set of FDs, this function will tell the user whether or not a specified FD is in the closure of the set of FDs. The input file must contain FDs for this function.

First, the function will display the FDs that are in the input file. Then, the toolbox will ask the user to input an FD. The user must input the FD in the same format as the FDs shown in the sample input files in Section A.6.2. That is, the FD must appear as follows and be followed by a period:

A – > B .

The toolbox will verify the format of the FD. If the FD is not in the proper format, the user is notified and asked if he wants to try again. If he does not want to try again, the toolbox returns to the FD Minimal Cover menu.

Once the user inputs a properly formatted FD, the toolbox determines if the FD is the closure of the set of FDs in the input file and then notifies the user of the results. The toolbox allows the user to try as many FDs with the current input file as desired, and then returns to the FD Minimal Cover menu.

A.6.4.4 Envelope Set for set of FDs/MVDs. An envelope set is the set of MVDs which is logically implied by a set of FDs and MVDs. The generated envelope set of MVDs can be used to decompose relations in the context of both FDs and MVDs.

This function will generate the envelope set of MVDs for a set of FDs and MVDs in a given input file. First, it displays the FDs and MVDs in the input file. Then, it displays the envelope set, and subsequently returns to the Utility Function menu.

A.6.4.5 Attribute closure. The closure of an attribute X with respect to a set of FDs is the set of all attributes functionally determined by X. This function computes the closure of one or more attributes and then displays the closure on the terminal.

First, the function displays the universal attributes, FDs and MVDs which are in the input file. Then, the function asks the user to input one or more attributes. Then finally, the function computes the closure of the attributes and displays the closure on the terminal.

The function allows the user to try as many sets of attributes with the current FDs as desired, and then returns to the Utility Function menu.

A.6.4.6 Dependency Basis. The dependency basis of an attribute is a set of sets of attributes which can be used to find the set of MVDs of the form $X \twoheadrightarrow Y$ logically implied by M. Given an input file which contains a set of universal attributes and a set of FDs and/or MVDs, this function will generate the dependency basis of a specified set of attributes.

This function follows the same format as the Attribute Closure function described above.

A.6.4.7 Instance of Armstrong Relation. Given an input file which contains a set of universal attributes and a set of FDs and MVDs, this function will generate an instance of an Armstrong relation. An Armstrong relation is a relation which satisfies precisely those dependencies in the input set, and no other 'accidental' dependencies.

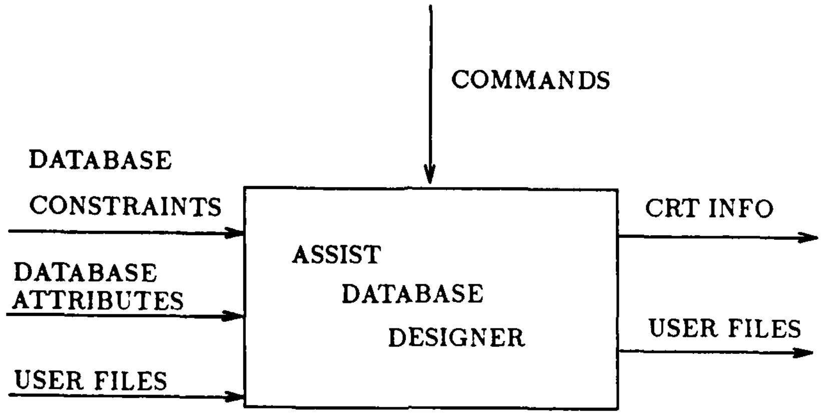
This function requires minimal interaction with the user. It simply notifies the user that the function has started, generates the instance of the Armstrong relation, and displays it on the terminal. The output is stored in a file which has the extension ". arm" preceded by the input file name. After the user has reviewed the output on the terminal, the toolbox returns to the Utility Function menu.

Appendix B. *SADT Diagrams*

B.1 Introduction

This Appendix contains the SADT diagrams which define and document the functional requirements for the Dependency Theory Toolbox. In the following pages of this appendix, each diagram is presented, and then the pages of text which immediately follow each diagram provide more information about the requirements conveyed in the diagram. The diagrams explain "what" the requirements are, and the pages of text which correspond to each diagram explain the requirements in more detail and explain "why" some of the requirements exist.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



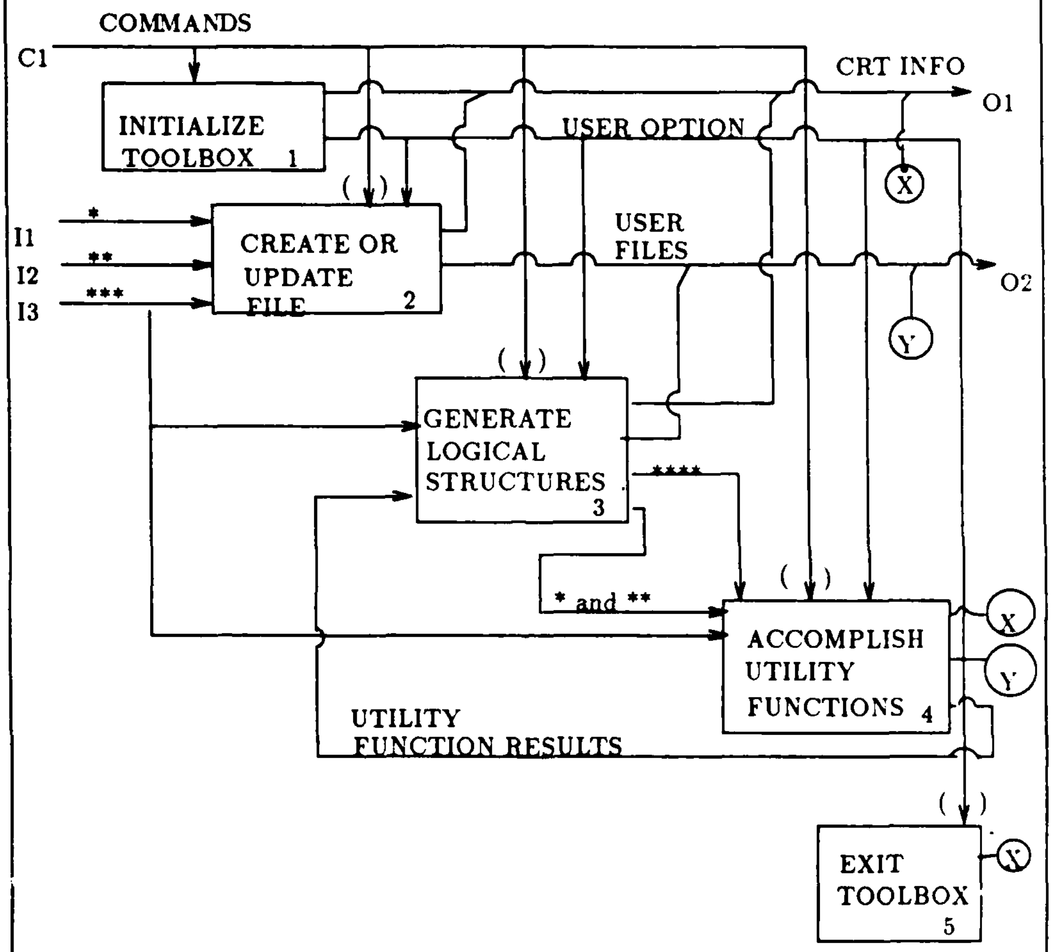
NODE: A-0	TITLE: Assist Database Designer	NUMBER: C-1

B.2 A-0 : Assist Database Designer

The activity, Assist Database Designer, encompasses the top level functional requirements for the Dependency Theory Toolbox. In general, the function of the toolbox is to assist the user with designing and studying the logical structures of relational databases. The activity allows the user to input DATABASE CONSTRAINTS (functional dependencies and multivalued dependencies) and DATABASE ATTRIBUTES interactively, or to specify existing USER FILES which contain the information. The user file can be created using the toolbox, or can be generated by other design tools in the prescribed format. The system is designed to accept existing files so that this toolbox can be used in conjunction with other database design tools developed at AFIT. For example, if the Entity-Relationship diagramming tool, which was developed by Mendez [22], stored the entities and attributes which were specified in the diagram in the proper format, the data file could be directly input into the toolbox.

The activity is controlled by COMMANDS which come from the user's terminal. The system communicates with the user by outputting CRT INFO on the user terminal and outputting other information into USER FILES which are stored on a system mass storage device.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



* - DATABASE CONSTRAINTS
** - DATABASE ATTRIBUTES
*** - USER FILES
**** - FUNCTION CALL

NODE: A0	TITLE: Assist Database Designer	NUMBER: C-2

B.3 A0 - Assist Database Designer

ABSTRACT: The activity, Assist Database Designer, provides the user with functions to help design and study the logical structures of relational databases. The activity is controlled by COMMANDS which come from the user's terminal. The system communicates with the user by outputting CRT INFO on the user terminal and outputting other data into USER FILES which are stored on a system mass storage device. The activity allows the user to input DATABASE CONSTRAINTS (functional dependencies and multivalued dependencies) and DATABASE ATTRIBUTES interactively, or to specify existing USER FILES which contain the information.

A1 - INITIALIZE TOOLBOX: This activity is automatically activated when the toolbox is started. The activity identifies the system and welcomes the user, and then provides the user with a menu of options. Once the user selects an option through COMMANDS from the user terminal, the activity calls the appropriate system function. This activity communicates with the user by outputting CRT INFO on the user terminal.

A2 - CREATE OR UPDATE DATABASE SPECIFICATION FILE: This activity interacts with the user to collect the DATABASE ATTRIBUTES and DATABASE CONSTRAINTS (FDs and MVDs), or gets the name of a previously defined file. The activity stores the database specifications in a properly formatted user file, or updates the previously defined file. The "properly formatted" user file is required mainly to provide a standard file which can be used to pass database specifications between this toolbox and other AFIT tools (e.g., the ER diagramming tool [22]), and as a place to store database specifications until needed by the system. The database specifications are stored in a user file on mass storage, and are available for input into the other activities shown on the A0 diagram as required.

A3 - GENERATE LOGICAL STRUCTURES: This activity generates the logical structures for a relational database by manipulating database attributes and constraints which are stored in a specified USER FILE. The USER FILE must be a file which contains database specifications in the standard format. Then, the selected option within this activity will call a routine to build the required data structures for the normalization process.

This activity can generate logical structures in 3NF or BCNF if the database constraints include functional dependencies (FDs), and 4NF if the constraints include a combination of both FDs and MVDs, or only multivalued dependencies (MVDs). The activity can either generate alternative logical structures, or generate a single design, depending on the user's preference. The activity communicates with the user by outputting CRT INFO on the user terminal, and stores database designs in USER FILES stored on mass storage. To accomplish the design tasks, the activity must call utility functions and pass these functions the appropriate database constraints and attributes. Then, the functions return the results to this activity for further processing.

A4 - ACCOMPLISH UTILITY FUNCTIONS: This activity provides many functions which can be accessed directly by the user or by the A3 activity. The user can select this activity from the main menu provided by the A1 activity, and then control it with COMMANDS input from the user terminal. The user must specify a USER FILE which contains database specifications in the standard format so the functions can build the appropriate data structures needed for the operations. Additionally, the A3 activity must call many of the functions provided by this activity in order to design logical structures. Once completed, the function returns the results to activity A3 for further processing.

A5 - EXIT TOOLBOX: This activity notifies the user of program termination and passes control back to the operating system.

B.4 A1 - INITIALIZE TOOLBOX

ABSTRACT: This activity is automatically activated when the toolbox is started. The activity identifies the system and welcomes the user, and then provides the user with a menu of options. Once the user selects an option through COMMANDS from the user terminal, the activity calls the appropriate system function. This activity communicates with the user by outputting CRT INFO on the user terminal.

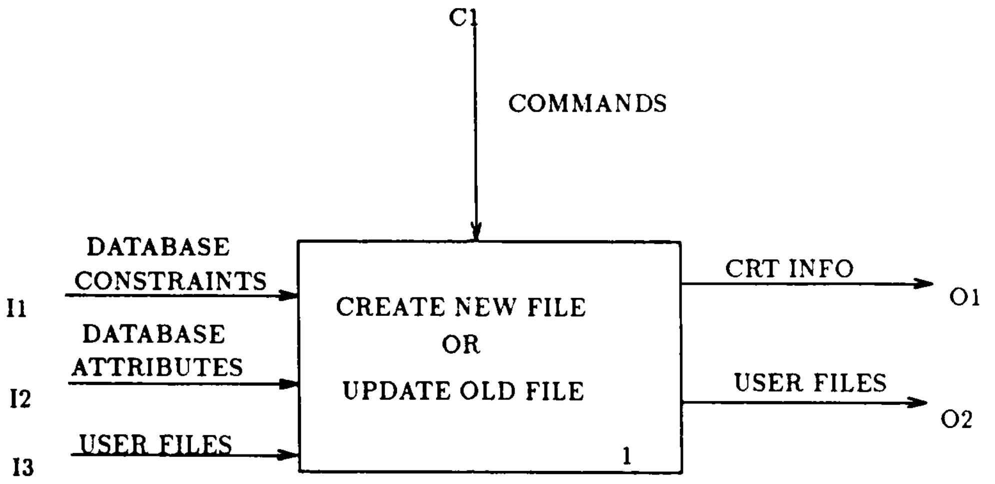
A1.1 - WELCOME USER: This activity generates a display to identify the toolbox and welcome the user. The initial display will remain on the screen until the user types a key to continue. After the user presses a key on the user terminal, this activity calls the PRESENT OPTIONS activity.

A1.2 - PRESENT OPTIONS: This activity will display a menu of the options the user can choose. The menu will contain the four main activities shown on the A0 diagram. The options will include the following:

1. Create or Update Database Specification File
2. Generate Logical Structures
3. Accomplish Utility Functions
4. Exit Toolbox

Once the user chooses an option, this activity will call the appropriate function.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



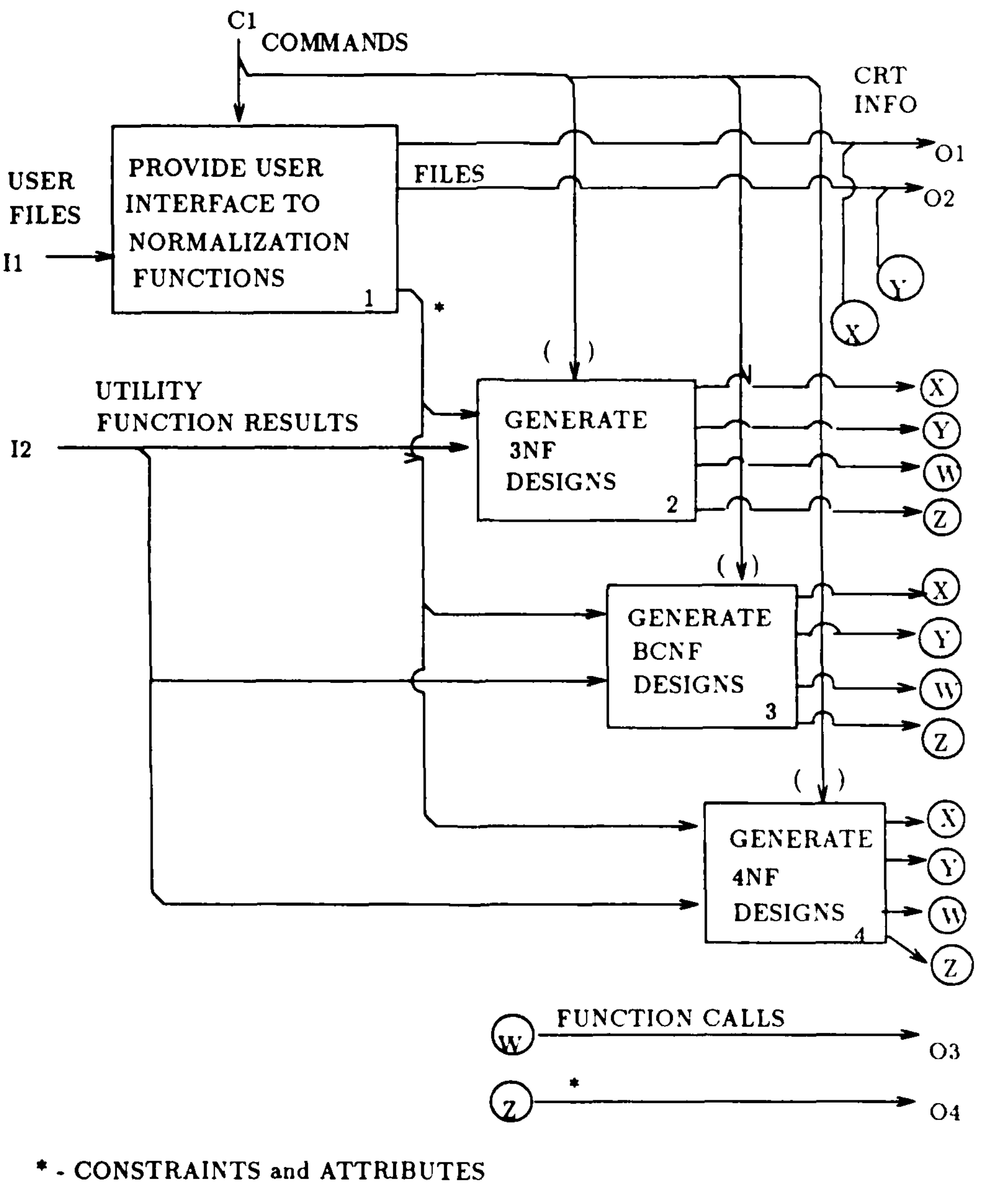
NODE: A2	TITLE: Create or Update Database Specification File	NUMBER: C-4

B.5 A2 - Create or Update Database Specification File

ABSTRACT: This activity interacts with the user to collect the DATABASE ATTRIBUTES and DATABASE CONSTRAINTS (FDs and MVDs), or gets the name of a previously defined file. The activity stores the database specifications in a properly formatted user file, or updates the previously defined file. The database specifications are stored in user files on mass storage, and are available for input into the other activities shown on the A0 diagram as required.

A2.1 - CREATE NEW FILE OR UPDATE OLD FILE: This activity is responsible for interacting with the user to collect the database specifications. The user will be expected to input the DATABASE ATTRIBUTES, and the FDs and MVDs (DATABASE CONSTRAINTS) required to hold over the given set of attributes in a specific format. Alternatively, the user may provide the name of a previously defined USER FILE which contains the information, or which contains part of the information and therefore needs to be updated. This activity then stores the properly formatted DATABASE SPECIFICATIONS into USER FILES. The USER FILES are stored on mass storage so they are available for other system functions. The user is informed of current activities by CRT INFO output to the user terminal.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



NODE: A3	TITLE: Generate Logical Structures	NUMBER: C-5

B.6 A3 - GENERATE LOGICAL STRUCTURES

ABSTRACT: This activity generates the logical structures for a relational database by manipulating database attributes and constraints which are provided in a specified USER FILE. The USER FILE must be a file which contains database specifications in the standard format. This activity can generate logical structures in 3NF or BCNF if the database constraints include functional dependencies (FDs), and 4NF if the constraints include only multivalued dependencies (MVDs) or a combination of both FDs and MVDs. The activity can either generate alternative logical structures, or generate a single design, depending on the user's preference. The activity communicates with the user by outputting CRT INFO on the user terminal, and stores database designs in USER FILES stored on mass storage. To accomplish the design tasks, the activity must call utility functions and pass these functions the appropriate database constraints and attributes. Then, the functions return the results to this activity for further processing.

A3.1 - PROVIDE USER INTERFACE TO NORMALIZATION FUNCTIONS: This activity displays the available options and obtains the name of the USER FILE which contains the database specifications in the standard format. The available options will include:

1. GENERATE 3NF DESIGNS
2. GENERATE BCNF DESIGNS
3. GENERATE 4NF DESIGNS

Once an option is selected, this activity calls the appropriate program function.

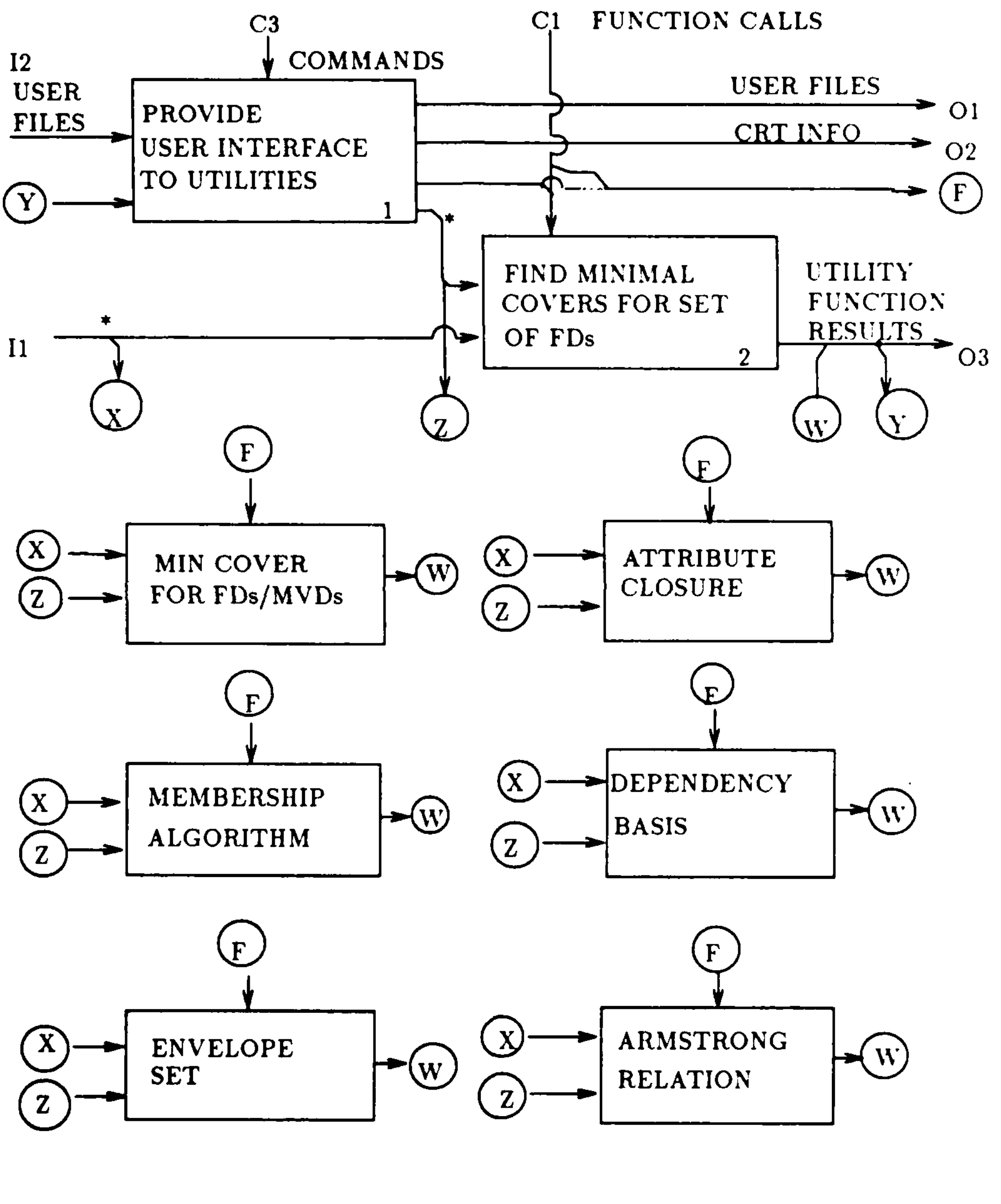
A3.2 - GENERATE 3NF DESIGNS: This activity generates logical structures in 3NF. The activity can either generate alternative logical structures, or generate a single design, depending on the user's preference. To accomplish normalization, this activity will use the database attributes and

FDs only from the database constraints provided in the USER FILE. That is, MVDs will not be used by the 3NF design algorithm to generate logical structures. The activity will store the logical structures in USER FILES, and will display them on the user terminal. Additionally, to accomplish the normalization process, this activity will call certain functions in the A4 activity and pass them the pertinent CONSTRAINTS AND ATTRIBUTES required by the functions. Then, this activity will use the UTILITY FUNCTION RESULTS to complete the normalization process.

A3.3 - GENERATE BCNF DESIGNS: This activity generates logical structures in BCNF. The functional requirements and capabilities described above for activity A3.2 also apply to this activity.

A3.4 - GENERATE 4NF DESIGNS: This activity generates logical structures in 4NF. To accomplish normalization, this activity will use the database attributes, and will use MVDs only from the database constraints provided in the database specifications in the USER FILE, or can use FDs and MVDs together, depending on the content of the USER FILE. Other than this difference, the functional requirements and capabilities described above for activity A3.2 also apply to this activity.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



* - CONSTRAINTS and ATTRIBUTES

NODE: A4	TITLE: Accomplish Utility Functions	NUMBER: C-6

B.7 A4 - ACCOMPLISH UTILITY FUNCTIONS

ABSTRACT: This activity provides many functions which can be accessed directly by the user or by the GENERATE LOGICAL STRUCTURES (A3) activity. The user can select this activity from the main menu provided by the INITIALIZE TOOLBOX (A1) activity, and then can control it with COMMANDS input from the user terminal. The user must specify a USER FILE which contains database specifications in the standard format. Then, the selected function will call a routine to build any unique data structures required for the particular operations. Additionally, the A3 activity must call many of the functions provided by this activity in order to design logical structures. Once completed, the function returns the results to activity A3 for further processing.

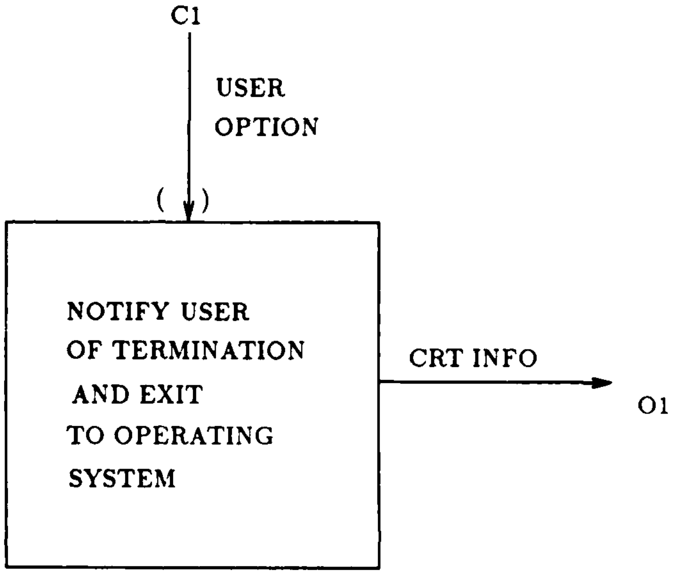
A4.1 - PROVIDE USER INTERFACE TO UTILITIES: This activity is activated if the user chooses to access the utility functions directly. The activity displays the available functions and obtains the name of the USER FILE which contains the database specifications in the standard format. The available functions will include:

1. Find minimal covers for a set of functional dependencies (FDs)
2. Find minimal covers for a set of FDs and multivalued dependencies (MVDs)
3. Membership algorithm (i.e., determine if an FD is in the closure of a set of FDs)
4. Find envelope set for a set of FDs and MVDs
5. Compute attribute closure
6. Find dependency basis of a set of attributes
7. Generate instance of an Armstrong relation

Once an option is selected, this activity calls the appropriate program function, and passes the CONSTRAINTS AND ATTRIBUTES.

A4.2 - A4.12: Each utility function requires that the database CONSTRAINTS AND ATTRIBUTES be provided as input. If the function has been called by activity A4.1, then the UTILITY FUNCTION RESULTS are passed back to that activity so they can be presented to the user. If the function has been called by one of the normalization functions in activity A3, then the results are passed back for further processing. The functions are controlled by function calls from either the user interface of activity A4.1 or the normalization tools in activity A3.

AUTHOR: Capt Stansberry	DATE: 18Dec87
PROJECT: Dependency Theory Toolbox	REV: 1.0



NODE: A5	TITLE: EXIT TOOLBOX	NUMBER: C-7

B.8 A5 - EXIT TOOLBOX

ABSTRACT: This activity notifies the user of program termination and exits to the operating system.

A5.1 - CLOSE FILES: This activity will check to see if any files are currently open, and if so will close them.

A5.2 - NOTIFY USER OF TERMINATION AND EXIT TO THE OPERATING SYSTEM: This activity notifies the user that the program has terminated and returns control to the operating system.

Appendix C. *Structure Charts*

The following structure charts show the top-level modules of the toolbox and how they are interrelatd. The charts were developed based on the SADT diagrams that were defined during the Requirements Analysis phase of the toolbox development.

The chart in Figure 17 shows the modules which are called by the main program driver. Then, the structure charts in Figures 18-28 show the modules that are called at the next lower level, and the information which is passed between them.

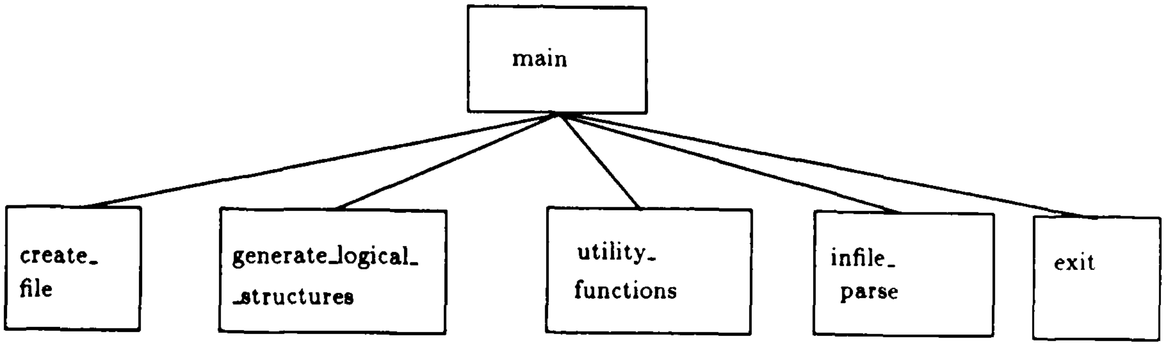


Figure 17. main structure chart

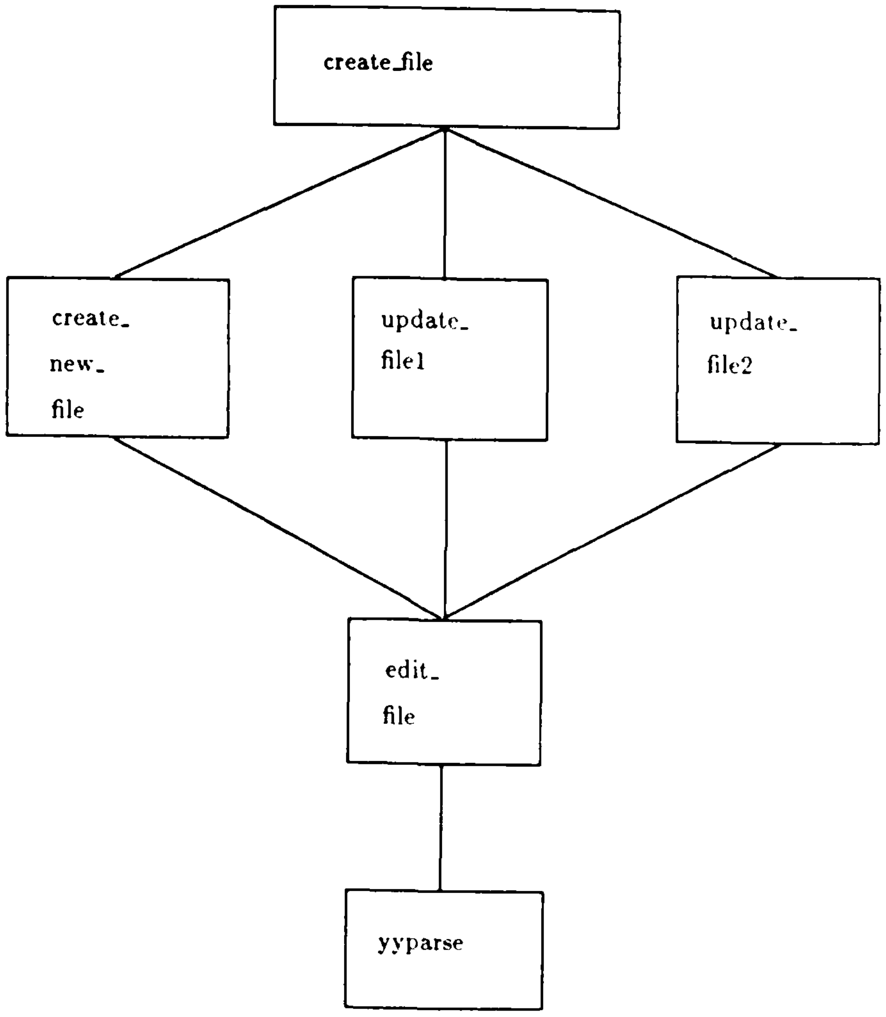


Figure 18. create_file structure chart

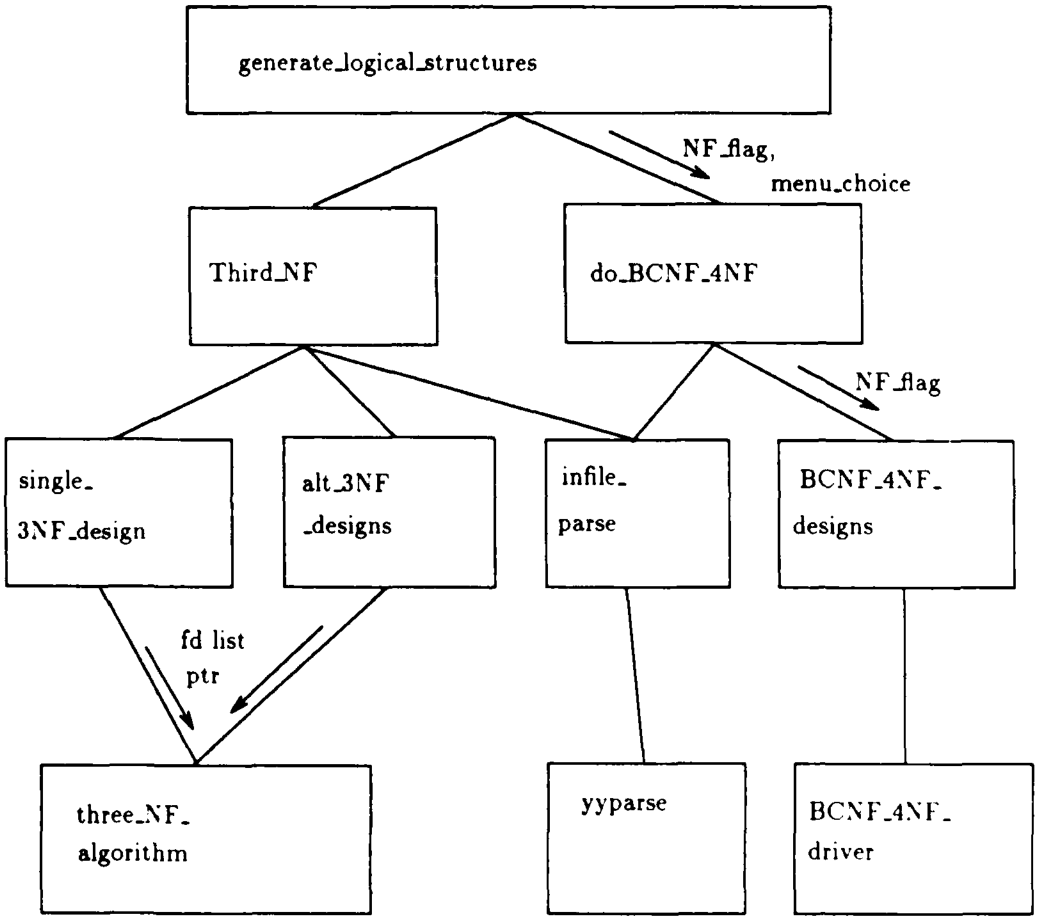


Figure 19. generate_logical_structures structure chart

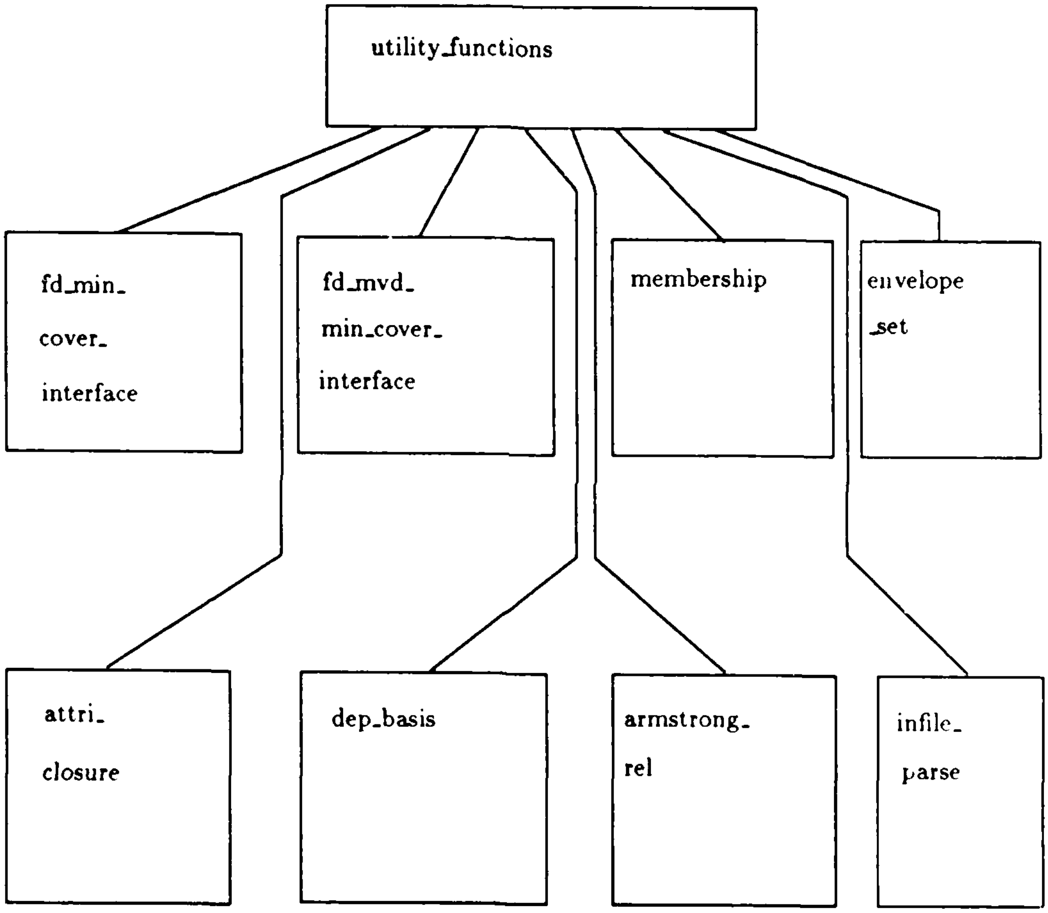


Figure 20. utility_functions structure chart

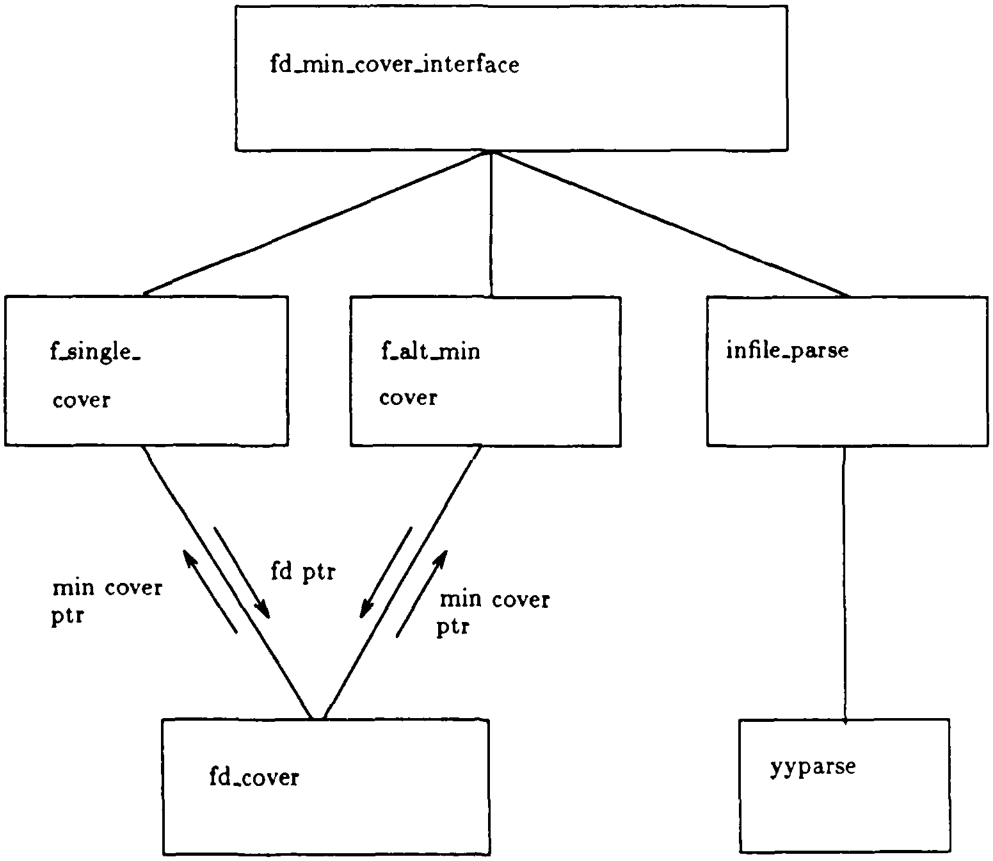


Figure 21. fd_min_cover_interface structure chart

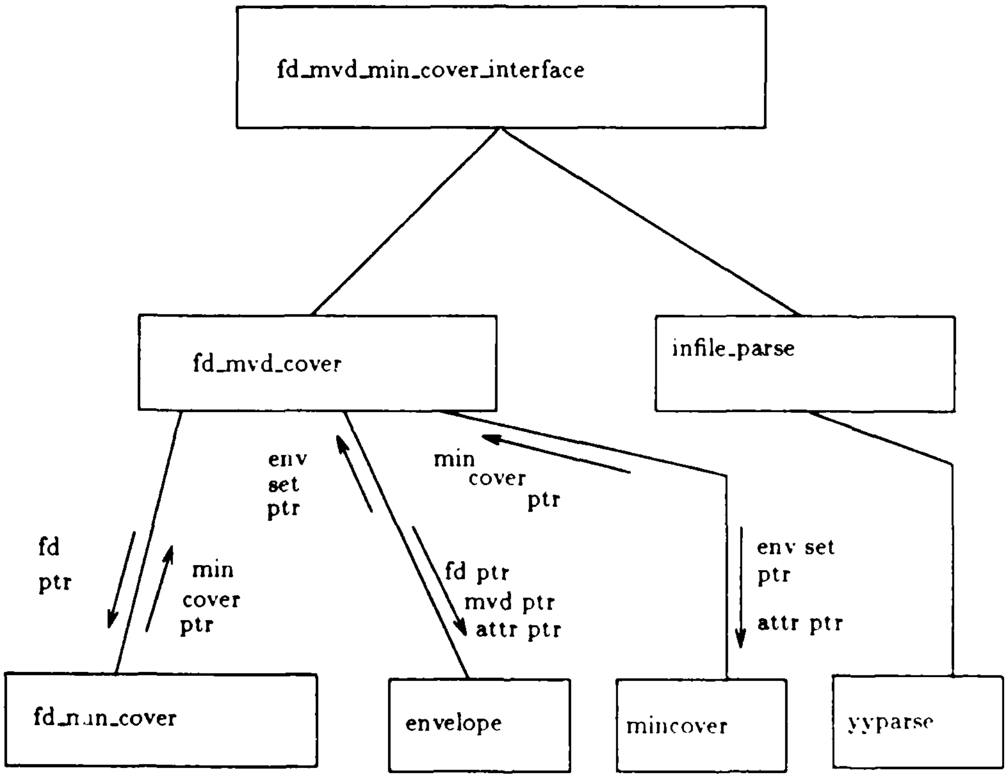


Figure 22. fd_mvd_min_cover_interface structure chart

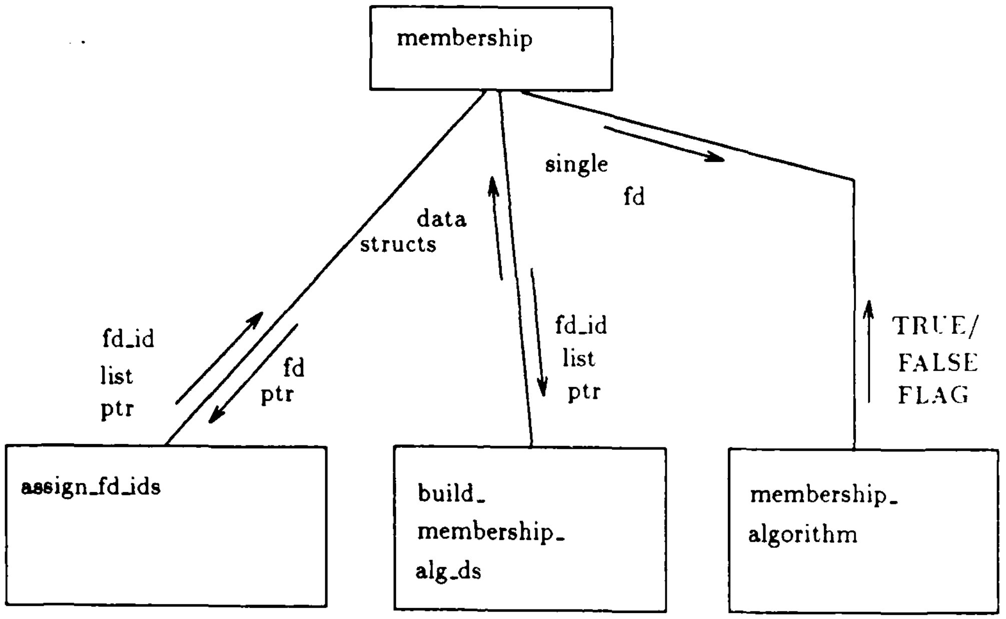


Figure 23. membership structure chart

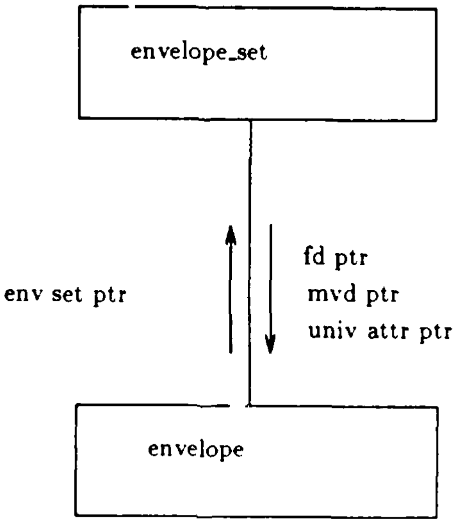


Figure 24. envelope_set structure chart

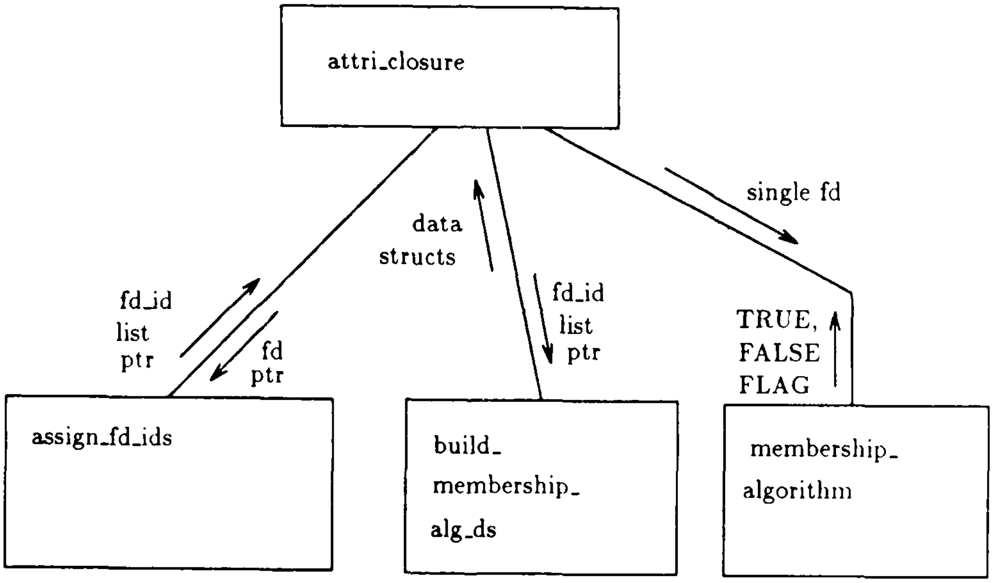


Figure 25. attri_closure structure chart

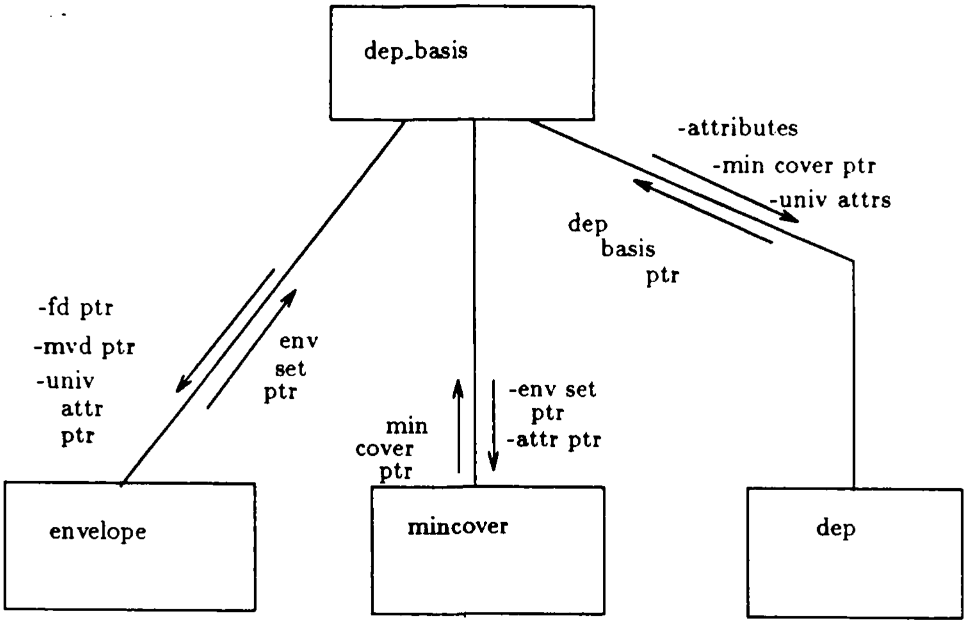


Figure 26. dep_basis structure chart

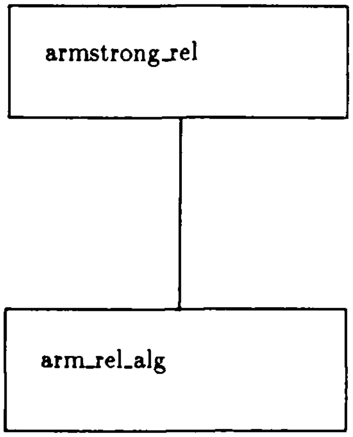


Figure 27. armstrong_rel structure chart

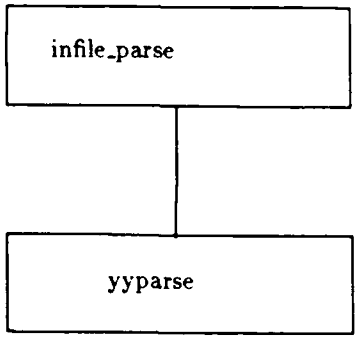


Figure 28. infile_parse structure chart

Appendix D. *Test Procedures and Input/Output Examples*

This Appendix contains the test procedures needed to test the accuracy of all toolbox functions. Each procedure specifies the required input and expected output for each test.

The input files which were used in the majority of the test procedures are presented below so they can be easily referenced in each test procedure. The only additional test files were required to test the Armstrong relation function, and therefore, they are presented in the Armstrong relation test procedure described below in Section D.11.

Input Files:

testfile1:

```
A B C D E F .

A -> DF .

AC ->-> B .

DE -> F .
```

testfile2:

```
A B C D .

A B -> D .

B -> C .

C -> D .
```

testfile3:

```
A B C D E F G H .

A B -> C .

C -> D .
```

```
D -> E .

DE -> FG .

FG -> H .
```

testfile4:

```
EMP_NUM SSN NAME SALARY .

SSN -> NAME SALARY EMP_NUM .

EMP_NUM -> NAME SSN SALARY .
```

testfile5:

```
boy girl bsl gsl date dance .

bsl ->-> boy .

gsl ->-> girl .

boy girl ->-> date .

boy girl ->-> dance .
```

testfile6:

```
A B C D E.

A ->-> E.

B C ->-> D .

B ->-> E .
```

testfile7:

```
A B C D .

A -> B .

B -> A .

C -> D .
```

D.1 Generate a single 3NF scheme

D.1.1 Test Case 1. **Input**: testfile2

Expected Output:

```
****************************************************

Single 3NF Design Output:

Minimal Cover:

B -> C

C -> D

3NF Schemes:

1.  Key = B

    Scheme = BC

2.  Key = C

    Scheme = CD

3.  Key = AB

    Scheme = AB
```

D.1.2 Test Case 2. **Input**: testfile3

Expected Output:

```
****************************************************

Single 3NF Design Output:


Minimal Cover:

C -> D

D -> E

D -> F
```

```
D -> G

AB -> C

FG -> H

3NF Schemes:

1.  Key = C

    Scheme = CD

2.  Key = D

    Scheme = DEFG

3.  Key = AB

    Scheme = ABC

4.  Key = FG

    Scheme = FGH
```

D.1.3 Test Case 3. **Input**: testfile4

Expected Output:

```
***************************************************

Single 3NF Design Output:


Minimal Cover:

EMP_NUM -> SSN

SSN -> EMP_NUM

SSN -> NAME

SSN -> SALARY

3NF Schemes:

1.  Key = EMP_NUM

    Scheme = EMP_NUM SSN
```

```
2.  Key = SSN

    Scheme = EMP_NUM SSN NAME SALARY
```

D.1.4 Test Case 4. **Input**: testfile5

Expected Output:

The system should display the following message:

> The input file does not contain any FDs, therefore, the universal set of attributes is by default in BCNF.

D.2 Generate alternative 3NF schemes

This function uses the same 3NF algorithm as tested above in the procedure for single 3NF designs, therefore, the output of the algorithm is already covered. Thus, the objective of this test procedure is simply to ensure the order of the FDs can be varied by the user. Therefore, several order changes were tried to verify that the function accurately changed the orders, and then produced the correct schemes.

D.3 Generate BCNF schemes

D.3.1 Test Case 1. **Input**: testfile2

Expected Output:

```
****************************************************

Single BCNF Design Output:


Envelope:

B ->-> A

C ->-> AB

Minimum Cover:
```

```
B ->-> A

C ->-> AB

Dependency Basis:

B ->-> A | CD

C ->-> D | AB

M-:

B ->-> A | CD

C ->-> D | AB

KEYs (sp-ordering) :

{C,B}

Schemes:

1. Key = C

   Scheme = CD

2. Key = B

   Scheme = BA

3. Key = B

   Scheme = BC
```

D.3.2 Test Case 2. **Input**: testfile1

Expected Output:

```
****************************************************

Single BCNF Design Output:

Envelope:

A ->-> BCE

AC ->-> B | E

DE ->-> ABC
```

```
Minimum Cover:

A ->-> BCE

AC ->-> E

DE ->-> ABC

Dependency Basis:

A ->-> DF | BCE

AC ->-> B | E | DF

DE ->-> F | ABC

M-:

A ->-> DF | BCE

AC ->-> B | E

DE ->-> F | ABC

KEYs (sp-ordering) :

{A,AC,DE}

Schemes:

1. Key = A

   Scheme = ADF

2. Key = AC

   Scheme = ACB

3. Key = AC

   Scheme = ACE
```

D.3.3 Test Case 3. **Input**: testfile7

Expected Output:

```
***************************************************

Single BCNF Design Output:
```

```
Envelope:

A ->-> CD

B ->-> CD

C ->-> AB

Minimum Cover:

A ->-> CD

B ->-> CD

C ->-> AB

Dependency Basis:

A ->-> B | CD

B ->-> A | CD

C ->-> D | AB

M-:

A ->-> B | CD

B ->-> A | CD

C ->-> D | AB

KEYs (sp-ordering) :

{A,B,C}

Schemes:

1. Key = A

   Scheme = AB

2. Key = C

   Scheme = CD

3. Key = C

   Scheme = CA
```

D.3.4 Test Case 4. **Input**: testfile6

Expected Output:

The system should display the following message:

> The input file does not contain any FDs, therefore, the universal set of attributes is by default in BCNF.

D.4 Generate 4NF schemes

D.4.1 Test Case 1. **Input**: testfile2

Expected Output:

```
**************************************************

Single 4NF Design Output:

Envelope:

B ->-> A

C ->-> AB

Minimum Cover:

B ->-> A

C ->-> AB

Dependency Basis:

B ->-> A | CD

C ->-> D | AB

M-:

B ->-> A | CD

C ->-> D | AB

KEYs (sp-ordering) :

{C,B}

Schemes:
```

```
1. Key = C

   Scheme = CD

2. Key = B

   Scheme = BA

3. Key = B

   Scheme = BC
```

D.4.2 Test Case 2. **Input**: testfile1

Expected Output:

```
****************************************************

Single 4NF Design Output:

Envelope:

A ->-> BCE

AC ->-> B | E

DE ->-> ABC

Minimum Cover:

A ->-> BCE

AC ->-> E

DE ->-> ABC

Dependency Basis:

A ->-> DF | BCE

AC ->-> B | E | DF

DE ->-> F | ABC

M-:

A ->-> DF | BCE

AC ->-> B | E
```

```
DE ->-> F | ABC

KEYs (sp-ordering) :

{A,AC,DE}

Schemes:

1. Key = A

   Scheme = ADF

2. Key = AC

   Scheme = ACB

3. Key = AC

   Scheme = ACE
```

D.4.3 Test Case 3. Input: testfile7

Expected Output:

```
*****************************************************

Single 4NF Design Output:

Envelope:

A ->-> CD

B ->-> CD

C ->-> AB

Minimum Cover:

A ->-> CD

B ->-> CD

C ->-> AB

Dependency Basis:

A ->-> B | CD

B ->-> A | CD
```

```
C ->-> D | AB

M-:

A ->-> B | CD

B ->-> A | CD

C ->-> D | AB

KEYs (sp-ordering) :

{A,B,C}

Schemes:

1. Key = A

   Scheme = AB

2. Key = C

   Scheme = CD

3. Key = C

   Scheme = CA
```

D.4.4 Test Case 4. **Input**: testfile6

Expected Outfile:

```
****************************************************

Single 4NF Design Output:

Envelope:

A ->-> E

B ->-> E

BC ->-> D

Minimum Cover:

A ->-> E

B ->-> E
```

```
BC ->-> D

Dependency Basis:

A ->-> E | BCD

B ->-> E | ACD

BC ->-> A | D | E

M-:

A ->-> E | BCD

B ->-> E | ACD

BC ->-> A | D

KEYs (sp-ordering) :

{A,B,BC}

Schemes:

1. Key = A

   Scheme = AE

2. Key = BC

   Scheme = BCA

3. Key = BC

   Scheme = BCD
```

D.5 Find minimal covers for set of FDs

D.5.1 Test Case 1. **Input**: testfile2

Expected Output:

Minimal Cover:

B – > C

C – > D

D.5.2 Test Case 2. **Input**: testfile3

Expected Output:

Minimal Cover:

C - > D

D - > E

D - > F

D - > G

AB - > C

FG - > H

D.5.3 Test Case 3. **Input**: testfile4

Expected Output:

Minimal Cover:

EMP_NUM - > SSN

SSN - > EMP_NUM

SSN - > NAME

SSN - > SALARY

D.5.4 Test Case 4. **Input**: testfile5

Expected Output:

The system should display the following message:

> **ERROR. the minimal cover algorithm cannot function with out functional dependencies.**

D.6 Find minimal covers for set of FDs and MVDs

D.6.1 Test Case 1. **Input**: testfile2

Expected Output:

Minimal Cover:

B − > C

C − > D

D.6.2 Test Case 2. **Input**: testfile6

Expected Output:

Minimal Cover:

A − > − > E

B − > − > E

BC − > − > D

D.6.3 Test Case 3. **Input**: testfile1

Expected Output:

Minimal Cover:

A − > − > BCE

ACE − > − > E

DE − > − > ABC

D.7 Membership algorithm

Input: Specify testfile2 as the input file and then follow the following interactive procedure.

Input "B – > D ." when the system displays the request:

> Please input an FD, and the function will determine if it is in the closure of the FDs.
> NOTE: the FD must be followed by a period.

Expected Output:

The system will display the message:

> The dependency is in the closure

Then the system will ask:

> Would you like to try another FD with this same set of FDs (y/n)?

Type "y" and the system will ask for another FD.

Input: "AB – > C ."

Expected Output:

The system will display the message:

> The dependency is in the closure

When the system asks if you want to do another FD, type "y", then:

Input: "C – > B ."

Output:

The system will display the message:

> The dependency is not in the closure

To end the function, respond "n" when the system asks if you want to do another FD with this set of dependencies.

D.8 Find the envelope set for a set of FDs and MVDs

D.8.1 Test Case 1. **Input**: testfile2

Expected Output:

Envelope Set:

B ->-> A

C ->-> AB

D.8.2 Test Case 2. **Input**: testfile6

Expected Output:

Envelope Set:

A ->-> E

B ->-> E

BC ->-> D

D.8.3 Test Case 3. **Input**: testfile1

Expected Output:

Envelope Set:

A ->-> BCE

AC ->-> B | E

DE ->-> ABC

D.9 Compute attribute closure

Input: Specify testfile2 as the input file and then follow the following interactive procedure.

Input "B" when the system displays the request:

Please input one or more attributes with at least one space between each attribute:

Expected Output:

The system will display the message:

```
The closure is:
B - > BCD
```

Then the system will ask:

Would you like to try another attribute with this same set of FDs (y/n)?

Type "y" and the system will ask for another set of attributes.

Input: "A B"

Expected Output:

The system will display the message:

```
The closure is:
AB - > ABCD
```

When the system asks if you want to do another set of attributes, type "y", then:

Input: "D"

Expected Output:

The system will display the message:

```
The closure is:
D - > D
```

To end the function, respond "n" when the system asks if you want to do another set of attributes with this set of dependencies.

D.10 Find dependency basis of set of attributes

Input: Specify testfilel as the input file and then follow the following interactive procedure.

Input "AC" when the system displays the request:

Please input one or more attributes with at least one space between each attribute:

Expected Output:

The system will display the message:

Dependency Basis:
$AC->->B|E|DF$

Then the system will ask:

Would you like to try another attribute with this same set of FDs (y/n)?

Type "y" and the system will ask for another set of attributes.

Input: "A"

Expected Output:

The system will display the message:

Dependency Basis:
$A->->DF|BCE$

When the system asks if you want to do another set of attributes, type "y", then:

Input: "F"

Expected Output:

The system will display the message:

Dependency Basis:
F – > – > ABCDE

To end the function, respond "n" when the system asks if you want to do another set of attributes with this set of dependencies.

D.11 Generate instance of an Armstrong relation

D.11.1 Test Case 1. **Input**:

```
A B C D .

A B -> D .

B -> C .

C -> D .
```

Expected Output:

```
           Armstrong Relation for file:  test_input2

 A   B   C   D

------------------------------------------------------

 A1  B1  C1  D1

 A1  B2  C2  D2

 A1  B3  C3  D1

 A1  B4  C1  D1

 A2  B1  C1  D1
```

D.11.2 Test Case 2. **Input**:

```
A B C D E F.

A -> B.

A ->-> C D.

A ->-> E F.

A C -> D.

E -> F.

E ->-> A B C D.
```

Expected Output:

```
           Armstrong Relation for file:  test_SM
 A   B   C   D   E   F

-------------------------------------------------------

 A1  B1  C1  D1  E1  F1

 A1  B1  C1  D1  E2  F2

 A1  B1  C1  D1  E3  F1

 A1  B1  C2  D2  E1  F1

 A1  B1  C2  D2  E2  F2

 A1  B1  C2  D2  E3  F1

 A1  B1  C3  D1  E1  F1

 A1  B1  C3  D1  E2  F2

 A1  B1  C3  D1  E3  F1

 A1  B1  C4  D1  E1  F1

 A1  B1  C4  D1  E2  F2

 A1  B1  C4  D1  E3  F1

 A2  B1  C1  D2  E1  F1

 A3  B2  C1  D1  E1  F1

 A4  B1  C1  D1  E1  F1
```

D.11.3 Test Case 3. **Input:**

```
A B C D E F G H I.

A -> B.

A -> D.

A -> C.

A -> E.
```

```
A -> F.

A -> G.

A ->-> H I.

B C D -> E.

B C D ->-> A F G H I.

F -> G.

F ->-> H I.

F ->-> A B C D E.

G -> F.

G ->-> H I.

G ->-> A B C D E.
```

Expected Output:

```
              Armstrong Relation for file:  test_ZM

 A   B   C   D   E   F   G   H   I

---------------------------------------------------------

 A1  B1  C1  D1  E1  F1  G1  H1  I1

 A3  B1  C1  D2  E2  F1  G1  H1  I1

 A4  B1  C1  D3  E1  F1  G1  H1  I1

 A5  B1  C2  D1  E2  F1  G1  H1  I1

 A6  B1  C3  D1  E1  F1  G1  H1  I1

 A7  B2  C1  D1  E2  F1  G1  H1  I1

 A8  B3  C1  D1  E1  F1  G1  H1  I1

 A9  B1  C1  D1  E1  F1  G1  H1  I1

 A1  B1  C1  D1  E1  F1  G1  H1  I2

 A3  B1  C1  D2  E2  F1  G1  H1  I2
```

```
A4  B1  C1  D3  E1  F1  G1  H1  I2

A5  B1  C2  D1  E2  F1  G1  H1  I2

A6  B1  C3  D1  E1  F1  G1  H1  I2

A7  B2  C1  D1  E2  F1  G1  H1  I2

A8  B3  C1  D1  E1  F1  G1  H1  I2

A9  B1  C1  D1  E1  F1  G1  H1  I2

A1  B1  C1  D1  E1  F1  G1  H2  I1

A3  B1  C1  D2  E2  F1  G1  H2  I1

A4  B1  C1  D3  E1  F1  G1  H2  I1

A5  B1  C2  D1  E2  F1  G1  H2  I1

A6  B1  C3  D1  E1  F1  G1  H2  I1

A7  B2  C1  D1  E2  F1  G1  H2  I1

A8  B3  C1  D1  E1  F1  G1  H2  I1

A9  B1  C1  D1  E1  F1  G1  H2  I1

A2  B1  C1  D1  E1  F2  G2  H1  I1
```

D.11.4 Test Case 4. **Input**:

```
A B C D.

A -> B C D.

B -> A.
```

Expected Output:

```
           Armstrong Relation for file:  test_arm

A   B   C   D

------------------------------------------------------

A1  B1  C1  D1
```

```
A2  B1  C1  D2

A3  B1  C1  D3

A4  B1  C1  D4

A5  B1  C2  D1

A6  B2  C1  D1
```

D.11.5 Test Case 5. **Input**:

```
A B C.

A B -> C.

C ->-> A.

C -> B.
```

Expected Output:

```
          Armstrong Relation for file:  test_comb2

 A   B   C

------------------------------------------------------

 A1  B1  C1

 A1  B2  C2

 A1  B3  C4

 A2  B1  C3

 A3  B1  C1
```

D.11.6 Test Case 6. **Input**:

```
A B .

A -> B.

B -> A.
```

Expected Output:

```
                Armstrong Relation for file:  test_one_to_one

 A   B

------------------------------------------------------

 A1  B1

 A2  B2
```

D.11.7 Test Case 7. **Input:**

```
A B C D.

A -> B.

B -> A.

C -> D.
```

Expected Output:

```
                Armstrong Relation for file:  test_1_to_1_2

 A    B    C    D

------------------------------------------------------

 A1   B1   C1   D1

 A1   B1   C2   D2

 A1   B1   C3   D1

 A2   B2   C1   D1
```

The results of the tests shown in the above test procedures are discussed in Chapter VI.

Bibliography

1. A. Albano et al. *Computer Aided Database Design, The DATAID Project.* Elsevier Science Publishers B. V., Amsterdam, 1985.
2. C. Batini et al. Database design activities within the dataid project. *Database Engineering*, 7:197–202, December 1984.
3. Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4:30–59, March 1979.
4. Catriel Beeri and Michael Kifer. An integrated approach to logical design of relational database schemes. *ACM Transactions on Database Systems*, 11:134–158, June 1986.
5. P. A. Bernstein. Synthesizing third-normal-form relations from functional dependencies. *ACM Transactions on Database Systems*, 1:277–298, December 1976.
6. Anders Bjornerstedt and Christer Hulten. Redl: a database design tool for the relational model of data. *Database Engineering*, 7:215–220, December 1984.
7. Michael L. Brodie. *Automating Database Design and Development: A SIGMOD 87 Tutorial.* Technical Report, Computer Corporation of America, Four Cambridge Center, Cambridge, MA 02142, May 1987.
8. S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of the ACM*, 29:524–546, June 1986.
9. Richard E. Cobb et al. The database designer's workbench. *Information Sciences*, 32:33–45, February 1984.
10. Robert M. Curtice. An automated logical data base design and structured analysis tool. *Database Engineering*, 7:221–226, December 1984.
11. Ronald Fagin. Armstrong databases. In *7th IBM Symposium on Mathematical Foundations of Computer Science*, pages 1–19, Kanagwa, Japan, 24-26 May 1982.
12. Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - a survey. *Proceedings of Symposia in Applied Mathematics*, 34:19–71, 1986.
13. I. T. Hawryszkiewycz. *Database Analysis and Design.* Science Research Associates, Chicago, 1984.
14. Marian Herman. A database design methodology for an integrated database environment. *Data Base*, 15:20–27, Fall 1983.
15. 2Lt Edward R. Jankus. *Development of Computer Aided Database Design and Maintenance Tools.* Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984.
16. Henry F. Korth and Abraham Silberschatz. *Database Systems Concepts.* McGraw-Hill, New York, 1986.
17. Mei Li. *A Nested Relational Database Design Tool.* Master's thesis, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland OH, May 1986.
18. Mary E. S. Loomis. *The Database Book.* Macmillan Publishing Company, New York, 1987.
19. Capt Thomas C. Mallary. *Design of the Human-Computer Interface for a Computer Aided Design Tool for the Normalization of Relations.* Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.
20. Michael A. Melkanoff. July 1987. In a July 1987 telephone conversation with Capt Stansberry, Dr. Melkanoff, at UCLA, indicated that the Armstrong relation tool is no longer available.

21. Michel A. Melkanoff and Carlo Zaniolo. Decomposition of relations and synthesis of entity-relationship diagrams. In Peter P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design*, pages 277–294, North-Holland Publishing Company, New York, December 1979.

22. Capt Ruben Mendez. *A Computer Aided Tool for Entity-Relationship Database Design*. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

23. Nancy E. Miller. *File Structures Using Pascal*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

24. David Reiner et al. The database design and evaluation workbench (ddew) project at cca. *Database Engineering*, 7:191–196, December 1984.

25. Mark A. Roth. *Theory of Non-First Normal Form Relational Databases*. PhD thesis, The University of Texas at Austin, Austin, Texas, May 1986.

26. A. M. Silva and M. A. Melkanoff. *Advances in Database Theory*. Volume 1, Plenum Publishing, New York, 1981.

27. Capt Charles T. Travis. *Interactive Automated System for Normalization of Relations*. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1983.

28. D. M. Tsou and P. C. Fischer. Decomposition of a relation scheme into boyce-codd normal form. *ACM-SIGACT*, 14:23–29, Summer 1982.

29. Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition edition, 1981.

30. Eric G. Vesely. *The Practitioner's Blueprint for Logical and Physical Database Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

31. S. Bing Yao et al. *Principles of Database Design*. Volume 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

32. Li-Yan Yuan and Z. Meral Ozsoyoglu. Logical design of relational database schemes. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 38–47, Association for Computing Machinery, March 23-25 1987.

33. Li-Yan Yuan and Z. Meral Ozsoyoglu. Unifying functional and multivalued dependencies for relational database design. In *Proceedings of the 5th ACM PODS*, pages 183–190, March 1986.

34. Carlo Zaniolo and Michel A. Melkanoff. On the design of relational database schemata. *ACM Transactions on Database Systems*, 6:1–47, March 1981.

Vita

Captain Charles W. Stansberry, Jr. was born on 23 August 1957 in Morgantown, West Virginia. He graduated from Beall High School in Frostburg, Maryland, in 1975 and attended the University of Maryland, from which he received the degree of Bachelor of Science in Biological Science in May 1980. Upon graduation, he received a commission in the USAF through the Officer Training School program. He completed the Communications Electronics Maintenance Officer technical training course at Keesler AFB, MS in July 1981, and completed the Communications Computer Programming technical training course in November 1981, also at Keesler AFB. He then served as a Computer Programmer/Analyst for the Real-time AUTODIN Interface and Distribution System (RAIDS) at the Air Force Communications Computer Programming Center (AFCCPC) at Tinker AFB, OK. Then, in May 1983 he became the Chief, DCT 9000 Programming Branch, also at AFCCPC (now named CCSO). In May 1984, he received the degree of Bachelor of Science in Data Processing from Central State University, Edmond, OK. His next assignment, in November 1984, was at Headquarters Space Command, Colorado Springs, CO, where he worked in the System Integration Office (SIO) as a Missile Warning System Communication Integration Officer until entering the School of Engineering, Air Force Institute of Technology, in June 1986.

Permanent address: 3658 Southbrook Drive
Beavercreek, Ohio 45430

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
AFIT/GCS/ENG/87D-26

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
School of Engineering

6b. OFFICE SYMBOL *(If applicable)*
AFIT/ENG

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS *(City, State, and ZIP Code)*
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

7b. ADDRESS *(City, State, and ZIP Code)*

8a. NAME OF FUNDING / SPONSORING ORGANIZATION

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
See Box 19

12. PERSONAL AUTHOR(S)
Charles W. Stansberry, B.S., Capt, USAF

13a. TYPE OF REPORT
MS Thesis

13b. TIME COVERED
FROM ________ TO ______

14. DATE OF REPORT *(Year, Month, Day)*
1987 December

15. PAGE COUNT
166

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
05	02	
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
COMPUTER AIDED DESIGN

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Title: DEVELOPMENT OF A DEPENDENCY THEORY TOOLBOX FOR DATABASE DESIGN

Thesis Advisor: Mark A. Roth, Captain, USAF

Approved for public release: IAW AFR 190-1.
Lynn Wolaver 31 Dec 87
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Mark A. Roth, Captain, USAF

22b. TELEPHONE *(Include Area Code)*
(513) 255-3576

22c. OFFICE SYMBOL
AFIT/ENG

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Much of the current dependency theory used to design and study relational databases exists in the form of published algorithms and theorems. However, hand simulating these algorithms can be a tedious and error prone chore. Therefore, the purpose of this thesis investigation was to design and implement a computer tool (that is, a "toolbox") which contains various relational database design algorithms and functions to help solve the problems created by hand simulating the algorithms.

This thesis includes a review of the activities typically done to design a relational database, and surveys the computer tools which are available, or are being developed, to assist database designers with the logical design of relational databases. The survey of computer tools indicated that although many researchers have developed computer tools to assist with relational database design, there are still many algorithms and functions which need to be incorporated into automated design tools.

The toolbox implements algorithms to accomplish the following functions: 3NF decomposition, 4NF decomposition, BCNF decomposition, envelope set, FD/MVD minimal cover, dependency basis, minimal cover, membership algorithm, attribute closure, Armstrong relation instance, and support for generation of alternative logical designs. A simple menu-driven interface was created to access the toolbox functions.

The toolbox is intended for use in an academic environment as a teaching aid and research tool rather than for practical application to database design problems. However, the tool could be used to design small relational databases which have a limited number of attributes. An evaluation of the toolbox done during the acceptance testing phase of development indicates that the tool can effectively serve in all of these capacities.

END

FILMED

MARCH, 1988

DTIC